Automator Programming Guide

Tiger

Contents

Tables, Figures, and Listings

Introduction to Automator Programming Guide

Automator is an application from Apple that automates repetitive procedures performed on a computer. With Automator users can construct arbitrarily complex workflows from modular units called actions. An action performs a discrete task, such as opening a file, cropping an image, or sending a message. A workflow is a number of actions in a particular sequence; when the workflow executes, data is piped from one action to the next until the desired result is achieved.

Apple includes a suite of ready-made actions with Automator, but developers are encouraged to contribute their own actions. You can create actions—which are implemented as loadable bundles—using either AppleScript, Objective-C, or a combination of the two languages.

Important: Automator was introduced in Mac OS X version 10.4. It does not run on earlier systems.

Who Should Read This Document

Any developer can create actions for Automator, as indeed can system administrators or "power users" who are familiar with AppleScript. But application developers have a particular motivation for developing actions. They can create actions that access the features of their applications, and then install these actions along with their applications. Users of Automator can then become aware of the applications and what they have to offer.

Developers can also contribute to Automator by making their applications scriptable or by providing a programmatic interface (via a framework) that developers use to create their actions.

Organization of This Document

Automator Programming Guide consists of the following articles:

- "Automator and the Developer" (page 9) describes what Automator can do and discusses the characteristics and types of actions,
- "How Automator Works" (page 15) gives an overview of the loadable-bundle architecture of Automator and the Objective-C classes of the Automator framework.
- "Design Guidelines for Actions" (page 23) lists guidelines for action development, including recommendations regarding I/O, naming, and the user interface.

- "Developing an Action" (page 29) guides you through the major steps required to develop an action.
- "Show When Run" (page 45) describes the Show When Run feature of actions and explains how you can customize it.
- "Implementing a Script-Based Action" (page 53) states the requirements for developing an action using AppleScript, suggests approaches you might take, and discusses hybrid actions—that is, actions based on both AppleScript and Objective-C code.
- "Implementing an Objective-C Action" (page 59) explains how to create an action implemented with Objective-C code.
- "Creating a Conversion Action" (page 67) describes what is required to create actions whose only purpose is to convert between types of input and output data.
- "Automator Action Property Reference" (page 69) defines the types and expected values for the Automator properties specified in an action's information property list (`Info.plist`).

See Also

- *Mac OS X Technology Overview* presents an overview of the Xcode integrated development environment.
- *AppleScript Overview* introduces the technology for creating and controlling scriptable applicatoins.
- *The Objective-C Programming Language* describes the Objective-C programming language and how you use it to create object-oriented programs.
- *Cocoa Bindings* discusses the Cocoa bindings mechanism and how you can incorporate it in your programs.

Automator and the Developer

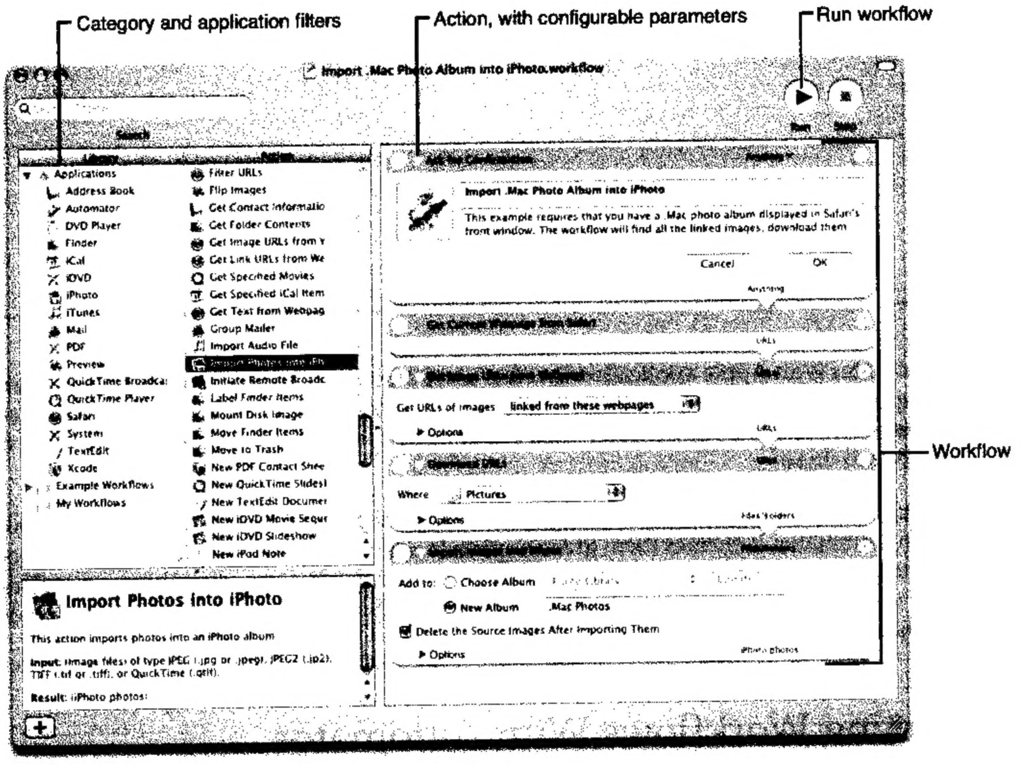
Automator is an Apple application that lets users automate repetitive procedures performed on a computer. With Automator, users can quickly construct arbitrarily complex sequences of configurable tasks that they can set in motion with a click of a button. And they do not need to write a single line of code to do so.

All kinds of Mac OS X users, including system administrators and developers, can derive huge benefits from Automator. It enables them to accomplish in a few seconds a complex or tedious procedure that manually might take many minutes. Developers can contribute to what Automator offers users in two ways: by making their applications scriptable and by creating loadable bundles specifically designed for Automator.

Constructing Workflows With Automator

As a developer, you can best appreciate how to integrate your software products with Automator by understanding how users might use the application. Figure 1 shows a typical Automator workflow.

Figure 1 A typical Automator workflow



Import .Mac Photo Album into iPhoto

This example requires that you have a .Mac photo album displayed in Safari's front window. The workflow will find all the linked images, download them

Get URLs of images: linked from these webpages

Where: Pictures

Add to: New Album .Mac Photos

[✓] Delete the Source Images After Importing Them

Import Photos into iPhoto

This action imports photos into an iPhoto album

Input: (image files) of type JPEG (.jpg or .jpeg), JPEG2 (.jp2), TIFF (.tif or .tiff), or QuickTime (.qtif).

Result: (iPhoto photos)

A procedure in Automator is known as a workflow, and a workflow consists of a sequence of discrete tasks called actions. An Automator action is a kind of functional building block. An Automator workflow hooks the actions together so that—in most cases—the output of one action is the input of the subsequent action. Clicking the run button in the upper-right corner of the window causes Automator to invoke each action of the workflow in turn, passing it (usually) the output of the previous action.

To create a workflow, users choose each action they want from the browser on the left side of the application window and drag it into the layout view in the right side of the window. They can search for specific actions by application, category, or keyword. Once dropped, an action presents a view which, in almost all cases, contains text fields, buttons, and other user-interface elements for configuring the action. In addition to configuring each action, users can move actions around in the workflow to put them in the desired sequence. They can then save the workflow as a document that can be run again and again in Automator.

Actions can be virtually anything that you can do on a Mac OS X system (version 10.4 and later). You can have an action that copies files, an action that crops an image, an action that burns a DVD, or an action that builds a project in Xcode. Actions can either interact with applications or draw on system resources to get their work done. By stringing actions together in a meaningful sequence—that is, in a workflow—a user can accomplish complex and highly customized procedures with a click of a button. Apple provides dozens of actions for use in Automator (Mac OS X version 10.4 and later).

Workflow Scenarios

A general description of Automator and its actions and workflows goes only so far to convey the power and flexibility of the application. To better appreciate what users can do with Automator—and how your actions can play a part in it—consider the following workflow scenarios:

- Workflow 1—Copies the content of all messages flagged in the currently displayed mailbox to a TextEdit document. This workflow consists of the following actions:
  1. Find Messages in Mail (Mail), configured to find messages whose flagged status is true
  2. Display Messages (Mail)
  3. Combine Messages (Mail)
  4. New TextEdit Document (TextEdit)
- Workflow 2—Imports images on a webpage (such as .Mac photo album) into iPhoto:

  This workflow requires users to go to the webpage in Safari before running the workflow.
  1. Get Current Webpage from Safari
  2. Get Image URLs from Webpage, configured to get the URLs linked from this webpage
  3. Download URLs, configured to a specified location on the local computer)
  4. Import Images into iPhoto, configured to put them in a new or existing album
- Workflow 3—Finds people with birthdays in the current month and creates an iCal event for each person:
  1. Find People with Birthdays (Address Book), configured to this month.
  2. New iCal Event, configured to create one event for each person received. It is also configured with any relevant event information, such as calendar and alarm.

Developing for Automator

Given an automation tool as powerful and flexible as Automator, it's clear that any action you develop for it benefits both you and the user, especially if that action accesses what your application has to offer. Users have an additional path to the features and services that make your products unique, and this in turn enhances the visibility and reputation of your products.

You don't even have to be a programmer (in the traditional sense) to develop actions for Automator. You can write an action in AppleScript as well as in Objective-C (and AppleScript actions have full access to such things as Standard Additions). You can also mix AppleScript scripts and Objective-C code when you develop an action; the script can invoke Objective-C methods implemented for the action. AppleScript has the advantage of being easier to learn, but it requires that applications that are the targets of commands be scriptable. As an object-oriented superset of ANSI C, Objective-C is a full-fledged programming language, which makes it harder to master than AppleScript but allows you to take full advantage of the Cocoa Objective-C frameworks—as well as all C-based frameworks.

In light of these language alternatives, there are three general types of Automator actions:

- An action that controls an application to get something done. AppleScript-based actions can do this if the application is scriptable. Objective-C actions can do this if the application has a public programmatic interface (as do, for example, Address Book and iChat).
- An action that uses system frameworks and other system resources to get something done. Actions that are strictly based on AppleScript cannot do this, but Objective-C actions—and actions that are a mixture of AppleScript and Objective-C—can. An Objective-C action, for example, could use the network-services classes NSNetService and NSNetServiceBrowser to implement an action that offers dynamic printer discovery.
- An action that performs some small but necessary "bridge" function such as prompting the user for input, writing output to disk, or informing the user of progress. You can create these types of actions using both AppleScript and Objective-C.

Important: Consider making your applications accessible to actions even if you do not provide any Automator actions of your own. For AppleScript-based actions, that means making your applications scriptable—that is, responsive to AppleScript commands. For Objective-C based actions, that means publishing a programmatic interface. Other developers may decide to create actions that access your application's services. See *AppleScript Overview* for information on making applications scriptable.

From a developer's perspective, actions have a common programmatic interface and a common set of properties. The programmatic interface is defined by three classes: AMAction, AMBundleAction, and AMAppleScriptAction (discussed in "How Automator Works" (page 15)). Action properties, which are specified in an action bundle's information property list (`Info.plist`), define characteristics such as:

- The types of data the action accepts and provides
- The application the action is associated with and the category of task it performs
- Other keywords to identify the action in searches
- The default parameters of the action
- Warnings given to users if the action, as configured, might result in loss of data

See "Automator Action Property Reference" (page 69) for further information on Automator action properties.

The Apple integrated development environment, Xcode, provides support for Automator action developers. There are Xcode project types for both AppleScript- and Objective-C based actions. Included in these project types are all required files and templates. For example, the default nib file (`main.nib`) contains an initial user interface, an instance of NSObjectController for establishing bindings, and File's Owner already set up to be your action object. For more on the Automator action project types see "Developing an Action" (page 29).

If you deploy actions as part of your product, be sure to install them in `/Library/Automator` (if they are to be available to all users of a system) or in `~/Library/Automator` (if they are to be available only to the current user). The actions in `/System/Library/Automator` are reserved for

Apple-developed actions. You can also install actions in the bundle of the application that is related to them. Automator looks for actions in application bundles as well. If you install actions, you may want to advertise them to your users so they can look forward to using them in new workflows.

How Automator Works

The following sections describe the development environment, runtime architecture, and class hierarchy for Automator actions.

The Development Components of Automator

The Xcode development environment integrates a number of technologies to make action programming as efficient as possible:

- Xcode project templates—Apple defines two project types for actions, one for AppleScript actions and another for Objective-C actions. Each project type contains all required files and resources, and supplies placeholder code and default configurations for the action-bundle project.
- AppleScript Studio—A powerful tool that helps you to create Mac OS X applications that support the Aqua user interface guidelines and that can execute AppleScript scripts.
- Interface Builder—A tool for creating graphical user interfaces from palettes of standard objects. Automator supplies its own palette (Cocoa-Automator) containing pop-up lists for files and folders. The action project template includes a preconfigured nib file.
- Cocoa bindings—Actions by default are designed to use the Cocoa bindings mechanism to communicate data automatically between an action's view and its internal parameters; an action's nib file is configured to support bindings.
- Objective-C frameworks—Support for Automator actions is implemented in Objective-C. Consequently, the Foundation, Application Kit, and Automator frameworks—all Objective-C frameworks—are automatically imported into and linked with action projects.

The final product of all these components, when an action is built, is a loadable bundle.

Loadable Bundle Architecture

The Automator application is based on a loadable bundle architecture. It loads loadable bundles called actions and executes the code they contain in the sequence determined by the current workflow, piping the flow of data from one action to the next.

Each action is packaged as a loadable bundle—or in the case of AppleScript-based actions, a potentially loadable bundle. A loadable bundle contains resources of various kinds and usually binary code but is not capable of executing that code on its own. The internal structure of a Cocoa bundle is in a form that an NSBundle object "understands." Using an NSBundle object, an application or framework can load the resources and code of a loadable bundle at runtime and integrate them with what it already contains. Loadable bundles are essentially a plug-in architecture.

When it launches, Automator immediately scans the currently installed action bundles and extracts from each bundle's information property list (`Info.plist`) the information necessary to display it and prepare it for use (see Figure 1). Automator actions (in the form of loadable bundles) are stored in standard file-system locations:

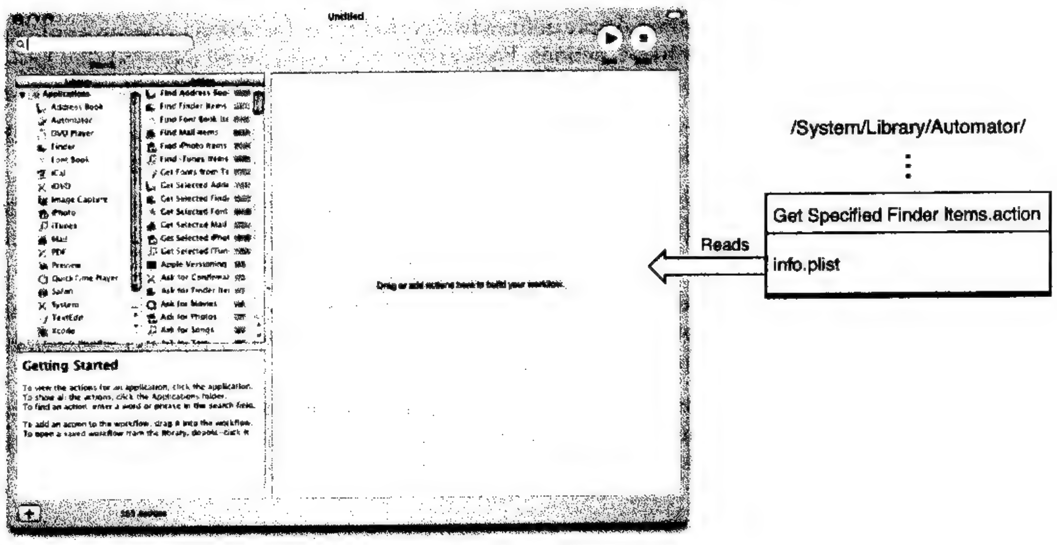
`/System/Library/Automator` — Apple-provided actions

`/Library/Automator` — third-party actions, general use

`~/Library/Automator` — third-party actions, private use

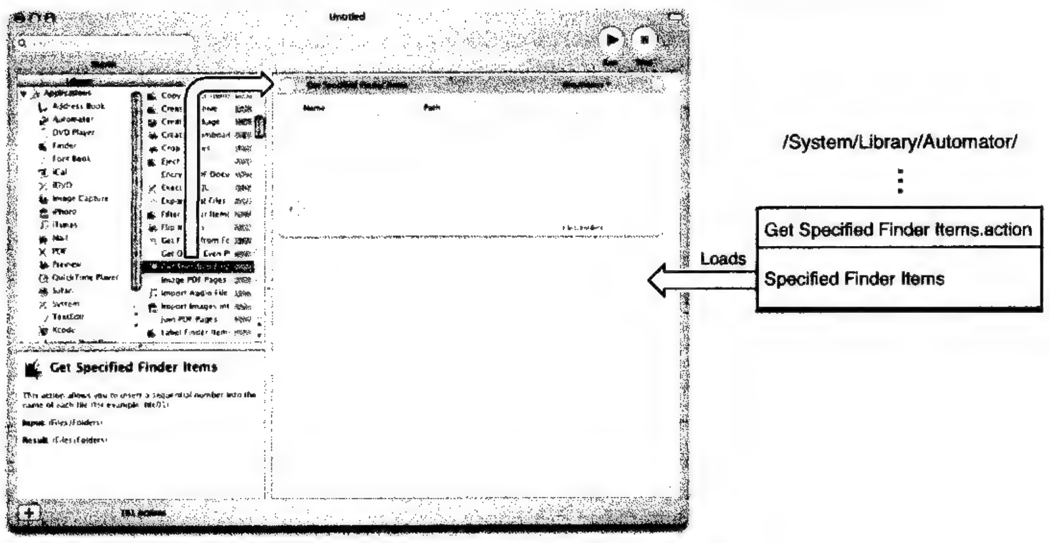
Automator also looks for actions that are stored inside the bundles of any registered applications. See "Testing, Debugging, and Installing the Action" (page 43) for information on installing actions in the bundles of their related applications.

Figure 1 When launched, Automator gets information about available actions



When it launches, Automator also loads any Mach-O code it finds in action bundles—which in this case are Objective-C actions only— resolving external references in the process. If the action is based on Objective-C, an instance of AMBundleAction (or a subclass of that class) is unarchived from the nib. However, if an action is purely AppleScript-based, Automator instead retains a reference to the bundle since there is no Mach-O code to load. When a user drags such an action into Automator's workflow area for the first time, the application dynamically generates code for the action from the script and loads it (see Figure 2).

Figure 2 When user drags AppleScript-based action into workflow, Automator loads bundle


Get Specified Finder Items
/System/Library/Automator/
Get Specified Finder Items.action
Specified Finder Items
Loads

Note: An action bundle is loaded when the action is first used when Automator launches. Thereafter, the action remains loaded until the user quits Automator.

The architectural details hereafter differ slightly based on whether the workflow is created for the first time or is unarchived.

A New Workflow

When the user creates a new workflow by dragging one or more actions into the workflow layout view, Automator does a couple of things:

- If the action is AppleScript-based it creates an instance of AMAppleScriptAction as the owner of the bundle, initializing it with an OSAScript object representing the compiled script.
- It gets the content view contained in the action's bundle and displays it within an action view in the workflow area, setting the default values of text fields and controls as defined by the action's `AMDefaultParameters` property.

Users modify the parameters of an action by choosing pop-up items, clicking buttons, entering text into text fields, and so on. When the workflow is ready, they click the run button to execute the workflow. With Automator acting as a coordinator, the application and each action perform the following steps in the workflow sequence:

- Automator invokes the `runWithInput:fromAction:error:` method of the action, passing it the output of the previous action as input.

  The AMAppleScriptAction class provides a default implementation of this method for AppleScript-based actions. This implementation calls the `on run` AppleScript handler.

- The settings made in the user interface of the action are propagated as parameters to the action object via Cocoa bindings.
- The action object (in most cases) takes the input and, based on the parameters, transforms it or does whatever its stated role is (such as importing it into an application or displaying output).
- As its last step, the action returns the result of its work (its output). If it does not affect the data passed it as input, it simply returns it unchanged.

While each action is busy, Automator displays a spinning progress indicator in the action view. If an error occurs in an action, the action view displays a red "X" and may display an error message. If the action successfully completes, its view displays a green checkmark. When the last action has finished its task, the workflow execution is over.

An Unarchived Workflow

When users save a workflow or copy one via the pasteboard (clipboard), the workflow and all of its actions are archived. Automator invokes the `writeToDictionary:` method of each of the actions in the workflow, passing in a mutable dictionary of the action's parameters and other information. The action may choose to modify the contents of the dictionary before returning from `writeToDictionary:`. Then the combined dictionaries of each action of the workflow are encoded and archived.

When Automator reads a workflow archive from disk or the pasteboard, it sends a `initWithDefinition:fromArchive:` message to each action in the workflow. In the first parameter of the message is a dictionary from which the action can re-create its state, particularly the last-selected parameters. The second parameter tells the newly created action object whether its definition is coming from an archive.

Once all actions in the workflows have been reinitialized from the archive, the workflow is ready to use. Users can select parameters in actions and run the workflow. Things continue on at this point as described in "A New Workflow" (page 17).

Threading Architecture

To improve runtime stability and give AppleScript-based actions access to resources such as Standard Additions, Automator has a threading architecture that puts different types of program activity on separate threads. Automator starts a workflow on a secondary thread (that is, a thread other than the main thread). But as it cycles through the actions of the workflow, it runs each action on a different thread depending on whether the action is based on AppleScript or Objective-C:

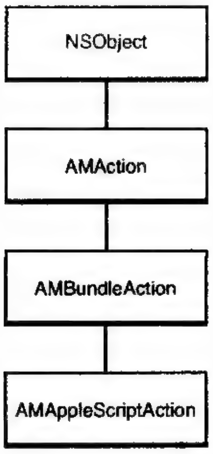
- If the action is based on AppleScript, it is executed on the main thread. The implementation permits users to cancel the execution of the action on this thread by clicking the stop button or pressing Command-. (period).
- If the action is based on Objective-C, it is executed on the secondary thread.

This threading architecture imposes restrictions on both writers of AppleScript-based and Objective-C actions. If an AppleScript-based action uses the `do shell script` command, the user interface is rendered unresponsive until the script completes; the only way users can cancel execution is to press Command-period. If Objective-C actions are to display windows, they must do so on the main thread using a method such as `performSelectorOnMainThread:withObject:waitUntilDone:`.

The Automator Classes

Automator as a technology includes not only the application and its actions but the Automator framework (`Automator.framework`). The framework implements much of the common behavior of actions and also provides a public interface defined by three classes: AMAction, AMBundleAction, and AMAppleScriptAction. These classes are hierarchically related (in terms of inheritance) as shown in Figure 3.

Figure 3 The Automator public classes



AMAction is an abstract class that specifies the interface and attributes essential to all actions. A major attribute of an action (as defined by AMAction) is its definition, a dictionary derived from the action properties specified in the bundle's information property list. The designated initializer of AMAction includes the action's definition in its signature: `initWithDefinition:fromArchive:` . The major method in an action's programmatic interface (as defined by AMAction) is `runWithInput:fromAction:error:`, which is briefly described in "Loadable Bundle Architecture" (page 15).

The AMBundleAction class directly inherits from AMAction and provides a concrete implementation of it. AMBundleAction defines the interface and common behavior of actions that are loadable bundles. AMBundleAction objects have three essential properties:

- An outlet connection to their associated view
- A reference to their bundle
- The parameters of the action—that is, the configuration choices users make in the action's user interface (stored in an NSDictionary object).

Bundled actions are designed to present a view, access the resources of the bundle, and access the parameters for the action. The implementation of the AMBundleAction class makes it possible for actions to be copied, pasted, and encoded for archiving.

AMAppleScriptAction is a subclass of AMBundleAction. It extends loadable action bundles so that AppleScript scripts can drive the action's logic instead of Objective-C code (although AppleScript and Objective-C code can be mixed in the implementation of an action). The sole outlet of AMAppleScriptAction is an OSAScript object representing the script; by default, this outlet is set to an object representing `main.applescript`.

You can create your own subclasses at the last two levels of the Automator class hierarchy to get objects with the characteristics and capabilities that you need. If you want to create loadable action bundles whose behavior is determined by scripting languages other than AppleScript (for example, Perl, shell scripting, or Python), you would subclass AMBundleAction.

Conceptually, the two fundamental external factors for an action are the input object passed it by the previous action (if any) and the parameters specified by users through the controls and text fields of the action's view. An instance of AMBundleAction accesses the input object in its implementation of `runWithInput:fromAction:error:` and obtains parameters directly from the user interface through the Cocoa bindings mechanism. For an AppleScript-based action (represented by an AMAppleScriptAction instance), the input object and parameters are even more explicit. They occur as the passed-in values in the `on run` handler, as shown in Figure 4.

Figure 4 A typical action script

```
on run {input, parameters}
    set output to input
    set filesToCopy to {}
    set toDirectory to |toDirectory| of parameters
    set replaceExisting to |replaceExisting| of parameters

    -- Check to see that we actually got some files to copy
    if (the class of input) is script then
        error my localized_string("No item references were passed to the Copy Files action.")
    end if

    -- Default the toDirectory to the desktop if one wasn't specified
    if the toDirectory is "" then
        set the toDirectory to "~/Desktop"
    end if

    -- Expand the toDirectory
    set toDirectory to «event appScalM» "stringByExpandingTildeInPath" given «class psof»:toDirectory

    -- See if the path exists and that it is in fact a directory
    If not («event appScalM» "directoryExistsAtPath:" given «class of C»:"CopyFilesAction", «class witQ»:toDirectory)
        as boolean then
        error my localized_string("The chosen folder was not found on this computer.")
    end if

    -- Create an array of posix paths from the input
    repeat with i in input
        copy (POSIX path of i) to end of filesToCopy
    end repeat


    -- Do the actual copy
    set duplicatedFiles to «event appScalM» "copyFiles:toDirectory:replaceExisting:"
        given «class of C»:"CopyFilesAction", «class witP»:{filesToCopy, toDirectory, replaceExisting}

    set output to {}
    repeat with i in duplicatedFiles
        copy POSIX file (i) as alias to end of output
    end repeat

    return output
end run

on localized_string(key_string)
    return «event appSlocS» key_string given «class iBWI»:"com.apple.Automator.CopyFiles"
end localized_string
```

For information that clarifies aspects of this scripting code, see "Implementing a Script-Based Action" (page 53)

Design Guidelines for Actions

As with other parts of the human interface in Mac OS X, actions should have a consistent look and feel so that users can easily use them. The following guidelines will help you achieve that consistent look and feel.

For concrete examples of well designed actions, look at the action project examples installed in `/Developer/Examples/Automator`.

What Makes a Good Action?

Perhaps the most important guideline is to keep your action as simple and focused as possible. An action should *not* attempt to do too much; by doing so it runs the risk of being too specialized to be useful in different contexts. An action should perform a narrowly defined task well. If an action you're working on seems like it's unwieldy, as if it's trying to do too much, consider breaking it into two or more actions. Small, discrete actions are better than large and complex actions that combine several different elements.

An action should inform the user what is going on, and if it encounters errors, it should tell users about any corrective steps that they might take. If an action takes a particularly long period to complete, consider displaying a determinate progress indicator. (Automator displays a circular indeterminate progress indicator when an action runs.)

You should provide an action in as many localizations as possible. See "Developing an Action" (page 29) for further information on internationalizing actions.

Action Input and Output

Interoperability is critical in the implementations of actions. An action's usefulness is limited by the types of data it can accept from other actions and give to other actions in a workflow. You specify these data types in the `AMAccepts` and `AMProvide` properties of actions using Uniform Type Identifiers (UTIs). The following guidelines apply to action input and output.

- Make the types of data the action provides and accepts as specific as possible. For example, if an image is coming from iPhoto, use `com.apple.iphoto.photo-object` as the UTI identifier instead of `public.image`.

- Specify multiple accepted and provided types, unless that is not appropriate for the action.
- If your action doesn't require input, always be sure to pass the input through as output to the next action (by returning it).
- Ideally, an action should accept and provide a list (or array) of the specified types.

For more on the `AMAccepts` and `AMProvides` properties and the supported UTIs, see "Automator Action Property Reference" (page 69).

Naming an Action

The following guidelines apply to the names of actions:

- Use long, fully descriptive names (for example, Add Attachments to Front Mail Message).
- Start the name with a verb that specifies what the action does.
- Use plural objects in the name—actions should be able to handle multiple items, whether that be URLs or NSImage objects.
  - However, you may use the singular form if the action accepts only a single object (for example Add Date to File Names, where there can be only one date).
- Don't use "(s)" to indicate one or more objects (for example, Add Attachment(s). Use the plural form.

The User Interface of an Action

The user interface of an action should adhere to the following guidelines:

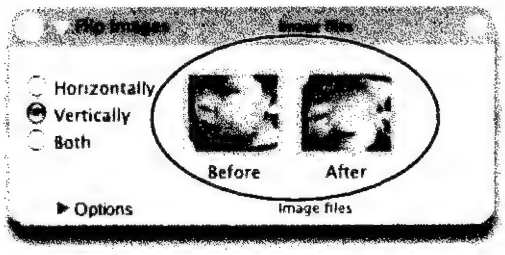
- Keep it simple:
  - Refrain from using boxes.
  - Minimize the use of vertical space; in particular, use pop-up lists instead of radio buttons even if there are only two choices.
  - Avoid tab views; instead, use hidden tab views to swap alternate sets of controls when users select a top-level choice.
  - Don't have labels repeat what's in the action title or description.
- Keep it small and consistent:
  - Use 10-pixel margins.
  - Use small controls and labels.
  - Give buttons the standard Aqua "Push Button" style.
  - Follow the Aqua guidelines (as suggested in Interface Builder).

- Implement behavior expected in a user interface—for example, tabbing between multiple text fields.

  If you do not know how to accomplish behavior like this using Interface Builder, see *Developing Cocoa Objective-C Applications: A Tutorial* or consult the *Interface Builder* documentation.

- Provide feedback and information to users:
  - Use determinate progress indicators when a user-interface element needs time to load its content; for example, an action that presents a list of current iCal calendars might need many seconds to load them.
  - Present examples of what the action will do where possible. For instance, the Make Sequential File Names action has an area of the view labeled "Example" that shows the effect of action options; the examples have the same font size and color as the rest of the action's user interface. Figure 1 shows an action that uses images for its example.

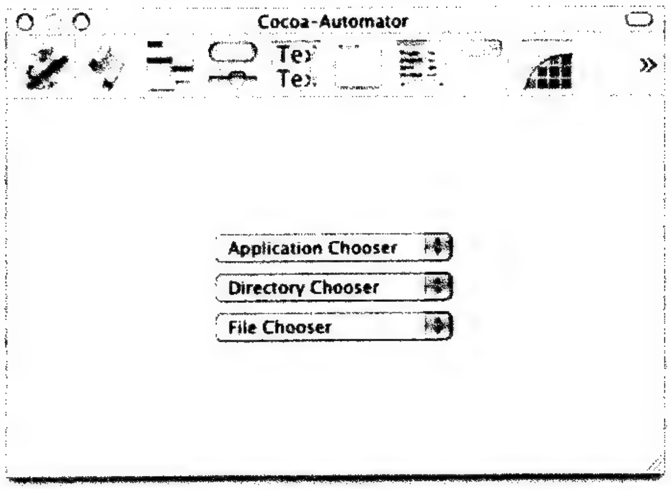
Figure 1 An action that includes an example of its effect



- Streamline file and folder selection and display:
  - Insert a pop-up list that includes standard file-system locations, such as Home, Startup Disk, Documents, Desktop, and so on.
  - The Cocoa-Automator palette (Figure 2) includes pre-configured pop-up lists for selecting applications, directories, and files; use these objects where appropriate.

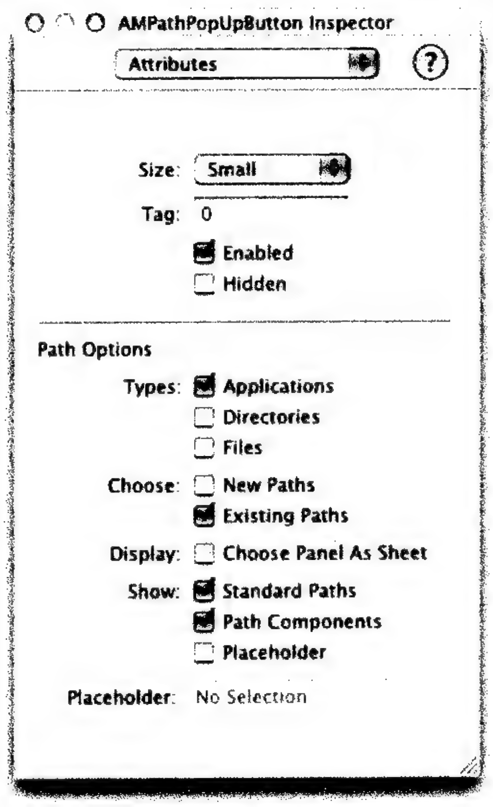
The Cocoa-Automator palette contains three specially implemented pop-up lists for presenting configurable lists of applications, directories, and files to users.

Figure 2 Cocoa-Automator palette



The Attributes pane in the Interface Builder inspector (Figure 3) allows you to configure these objects in various ways.

Figure 3 Attributes inspector for Cocoa-Automator palette objects



AMPathPopUpButton Inspector

Attributes

Size: Small

Tag: 0

[✓] Enabled

[ ] Hidden

Path Options

Types: [✓] Applications

[ ] Directories

[ ] Files

Choose: [ ] New Paths

[✓] Existing Paths

Display: [ ] Choose Panel As Sheet

Show: [✓] Standard Paths

[✓] Path Components

[ ] Placeholder

Placeholder: No Selection

Developing an Action

It's easy to develop an Automator action. Because an action is a loadable bundle, its scope is limited and hence the amount of code you need to write is limited. Apple also eases the path to developing an action because of all the resources it places at your disposal. When you create an action project, the Xcode development environment provides you with all the necessary files and settings to build an action. You just need to follow certain steps—described in this document—to arrive at the final product.

The steps for developing an action don't necessarily have to happen in the order given below. For example, you can write an action description at any time and you can specify the Automator properties at any time.

Creating an Automator Action Project

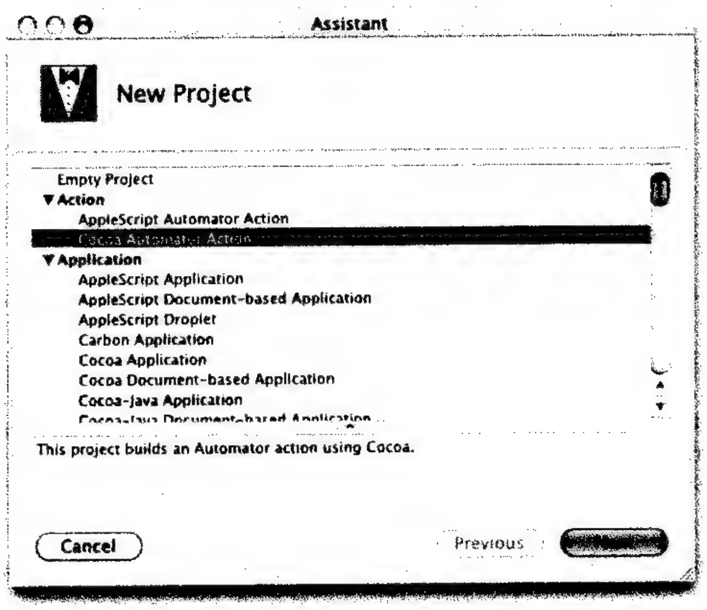
To create an Automator action project, launch the Xcode application and choose New Project from the File Menu. From the first window of the New Project assistant, select one of the two Automator action projects, depending on your language choice:

AppleScript Automator Action

Cocoa Automator Action

Figure 1 shows the selection of the Cocoa Automator action.

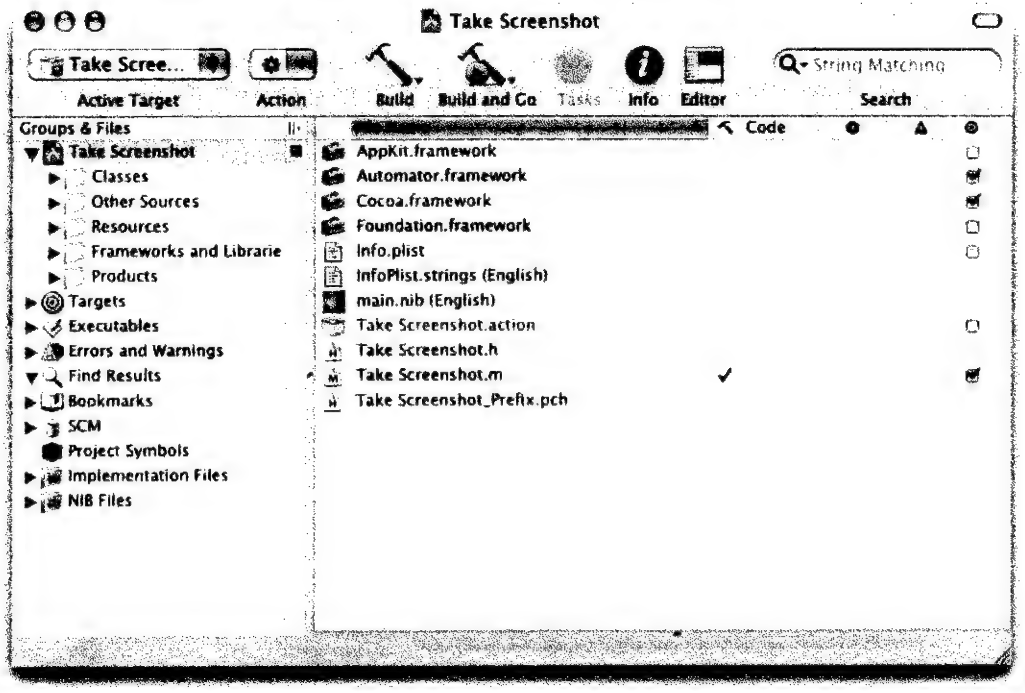
Figure 1 Choosing a Cocoa Automator Action project type in Xcode



This project builds an Automator action using Cocoa.

Complete the remaining window of the New Project assistant as usual—specify a file-system location for the project and enter a project name. When you finish, an Xcode window similar to the one in Figure 2 is displayed.

Figure 2 Template files for a Cocoa Automator Action project



You'll become more familiar with many of the files and file wrappers displayed here in the sections that follow. But here is a summary of the more significant items:

- `Automator.framework`—The project automatically adds the Automator framework to the project.
- `Info.plist`—The information property list for the bundle includes the Automator properties; comments inserted in the value elements provide helpful hints. See "Specifying Action Properties" (page 36) for further information.
- `projectName.h` and `projectName.m`—A Cocoa action project includes header and implementation template files prepared for an AMBundleAction subclass (including a null implementation of `runWithInput:fromAction:error:`).

  An AppleScript action project includes a template `main.applescript` file instead of Objective-C template files.
- `main.nib`—The bundle's nib file, partially prepared for the action. See "Constructing the User Interface" (page 31) for details.

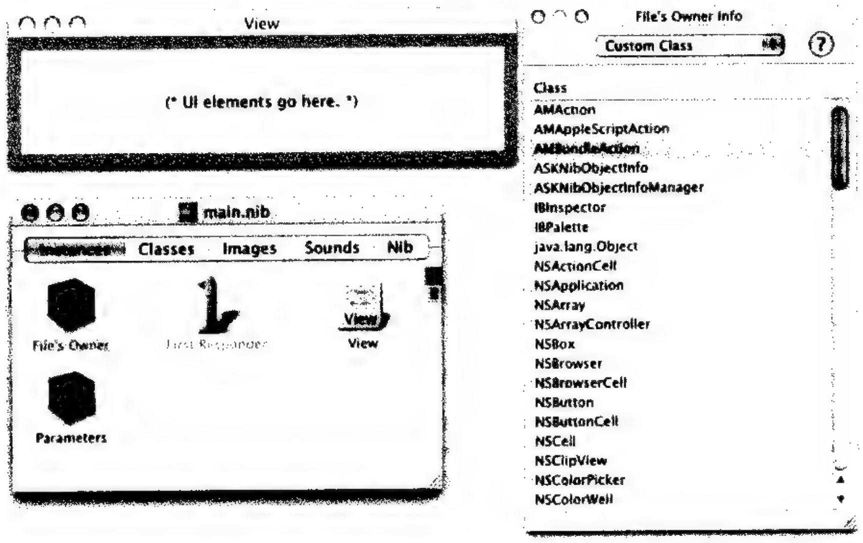
Constructing the User Interface

In the Xcode project browser, double-click `main.nib` to open the nib file of the action in Interface Builder. The nib file contains the usual File's Owner and First Responder instances, but it also contains two other items specific to actions:

- A blank NSView object (known as the content view of the action, or simply, action view) on which you set the controls, text fields, and other user-interface elements for setting the parameters of the action.
- An instance of the NSObjectController class named Parameters. The Parameters instance is used for establishing bindings between objects on the user interface and the action instance.
- For AppleScript-based actions, an AppleScript Info instance that AppleScript Studio uses to contain the names and event handlers for objects in the nib file.

Several objects and relationships in `main.nib` are already initialized for you. For example, File's Owner is set in the Custom Class pane of the Info window to be an instance of the AMBundleAction class (as shown in Figure 3). (If your project is for an AppleScript Automator action, File's Owner is set instead to AMAppleScriptAction.) With File's Owner still selected, if you choose the Connections pane of the Info window, you will see that the `view` outlet of the AMAction object has been connected to the action's view.

Figure 3 File's Owner set to an instance of AMBundleAction



Construct the user interface of the action as you would for any view in Interface Builder. However, keep in mind that actions, because they share space in an Automator workflow with other actions, should be compact, even minimal. Place only those controls, text fields, and other user-interface objects that are absolutely necessary. And try to be conservative in your use of vertical space; for example, prefer pop-up lists over radio buttons for presenting users with a choice among multiple options. For more on user-interface guidelines for actions, see "The User Interface of an Action" (page 24) in "Design Guidelines for Actions" (page 23).

Note: The Automator development environment includes an Interface Builder palette with user-interface objects specially designed for actions. This palette is named Cocoa-Automator. To load this palette, select the Palettes pane in the Interface Builder preferences, click Add, and select `AMPalette.palette` in `/Developer/Extra/Palettes`.

An important feature of Automator actions is Show When Run. This feature enables the users of workflows (as distinct from the *writers* of workflows) to set the parameters of actions when they run the workflow as an application. By default, actions have an Options disclosure button in their lower-left corner that, when enabled, exposes additional controls; these controls allow a workflow writer to select the parts of an action's view that are presented to users when the workflow is executed. Developers can customize the Show When Run feature; see "Show When Run" (page 45) for details.

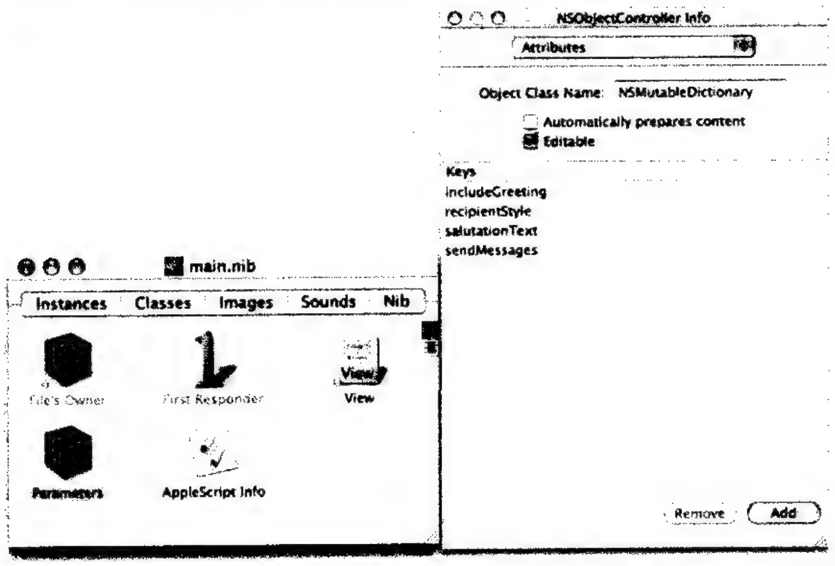
After constructing the action's view, establish the bindings between the objects on the user interface and the action object. Bindings are a Cocoa technology conceptually based in the Model-View-Controller paradigm. In Model-View-Controller, objects in a well-designed program assume one of three roles: view objects that present data and receive user input; model objects that contain data and operate on that data; and controller objects that mediate the transfer of data between view and model objects. The bindings mechanism automatically synchronizes the exchange of data between view and model objects, reducing the need for controller objects and all the "glue" code they usually entail. Automator actions use bindings to communicate data between objects in an action's view and the parameters dictionary (or record) that all actions have to record the settings users specify for an action.

The project templates for Automator actions are pre-configured to use bindings instead of the target-action and outlet mechanisms (for Cocoa-based actions) or event handlers specified in the AppleScript pane of Interface Builder for AppleScript-based actions. If you prefer, you may use these mechanisms along with the Automator framework's "update parameters" API to update an action's parameters manually. For information on this approach, see "Updating Non-bound Parameters" (page 57) (for AppleScript-based actions) or "Updating Action Parameters" (page 63) (for Objective-C actions).

The following steps summarize what you must do to establish the bindings of an action in Interface Builder. For a through description of the Cocoa bindings technology, see *Cocoa Bindings*.

1. Select the Parameters instance in the nib file window and choose the Attributes pane of the Info window. In the table labeled "Keys" add the keys for each user-interface object whose setting or value you need to access. See Figure 4 for an example.

Figure 4 Adding the keys of the Parameters instance (NSObjectController)



Object Class Name: NSMutableDictionary

Automatically prepares content

Editable

Keys
includeGreeting
recipientStyle
salutationText
sendMessages

You should use the same keys given here as keys and names elsewhere in the action implementation:

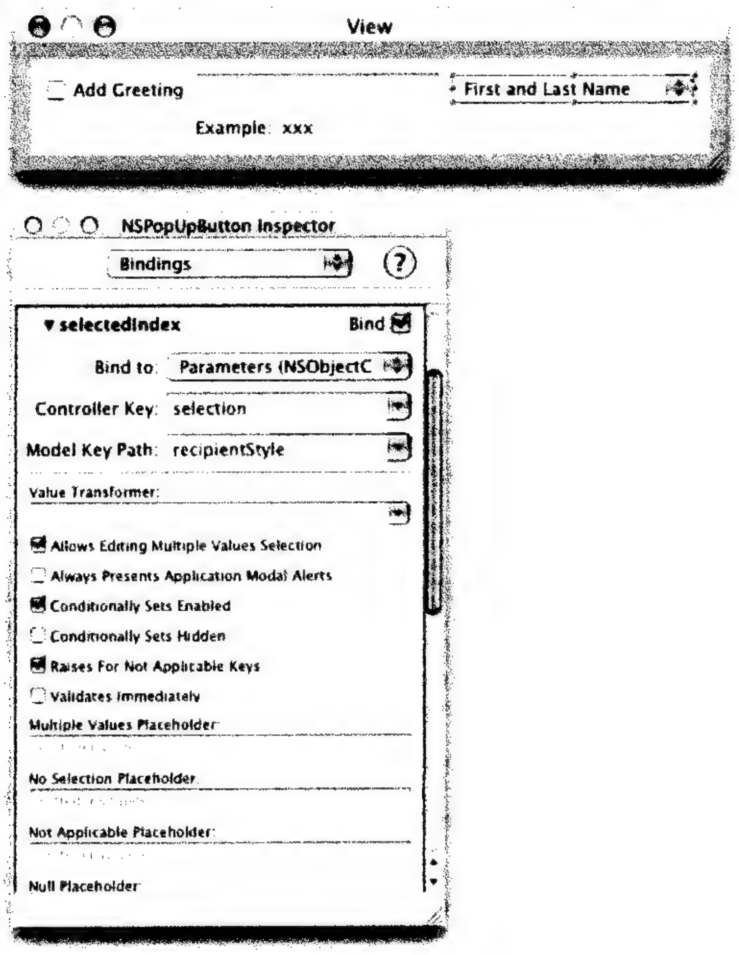
- As the keys for the `AMDefaultParameters` property of Automator. (The `AMDefaultParameters` property specifies the initial settings of an action; see "Specifying Action Properties" (page 36) and "Automator Action Property Reference" (page 69) for details.)
- As the names of outlet instance variables and accessor methods in Objective-C actions
- As the keys to the parameters record passed into an AppleScript-based action in its `on run` handler; see "The Structure of the on run Command Handler" (page 53) for further information.

2. With the Parameters instance still selected, choose the Bindings pane of the Info window and expand the "contentObject" subpane. Set the "Bind to" pop-up item to File's Owner and enter `parameters` in the "Model Key Path" field.

   The parameters key path refers to the `parameters` property defined by the AMBundleAction class—that is, the dictionary or record used to capture the settings users make in an action's view.

3. For each user-interface object of the action, establish a binding to the appropriate parameters key. For example, Figure 5 shows the binding for a pop-up list.

Figure 5 Establishing a binding for a pop-up list



View

☐ Add Greeting — First and Last Name

Example: xxx

NSPopUpButton Inspector

Bindings

▾ selectedIndex — Bind ✓

Bind to: Parameters (NSObjectC

Controller Key: selection

Model Key Path: recipientStyle

Value Transformer:

✓ Allows Editing Multiple Values Selection

☐ Always Presents Application Modal Alerts

✓ Conditionally Sets Enabled

☐ Conditionally Sets Hidden

✓ Raises For Not Applicable Keys

☐ Validates Immediately

Multiple Values Placeholder:

No Selection Placeholder:

Not Applicable Placeholder:

Null Placeholder:

The binding varies by type of user-interface object. For example, pop-up lists usually work off the index of the item (`selectedIndex`), checkboxes have a Boolean `value`, and text fields have a string `value`.

When you are finished with the action's user interface, save the nib file and return to the Xcode project.

Important: If you are deploying the action in multiple localizations, make sure you create a separate nib file and user interface for each localization. See "Internationalizing the Action" (page 40) for further information.

Specifying Action Properties

The Automator application uses special properties in an action's information property list (`Info.plist`) to get various pieces of information it needs for presenting and handling the action. This information includes:

- The name of the action
- The icon for the action
- The application, category, and keywords for the action
- The types of data the action accepts and the types of data it provides
- The description of the action

Automator properties have the prefix "AM". The project template for actions includes almost all of the properties you need (or may need) to specify. As shown in Figure 6, the template provides helpful comments as placeholders for key values. In creating an action, you need to supply real values for these keys. "Automator Action Property Reference" (page 69) describes the Automator properties, including their purpose, value types, and representative values.

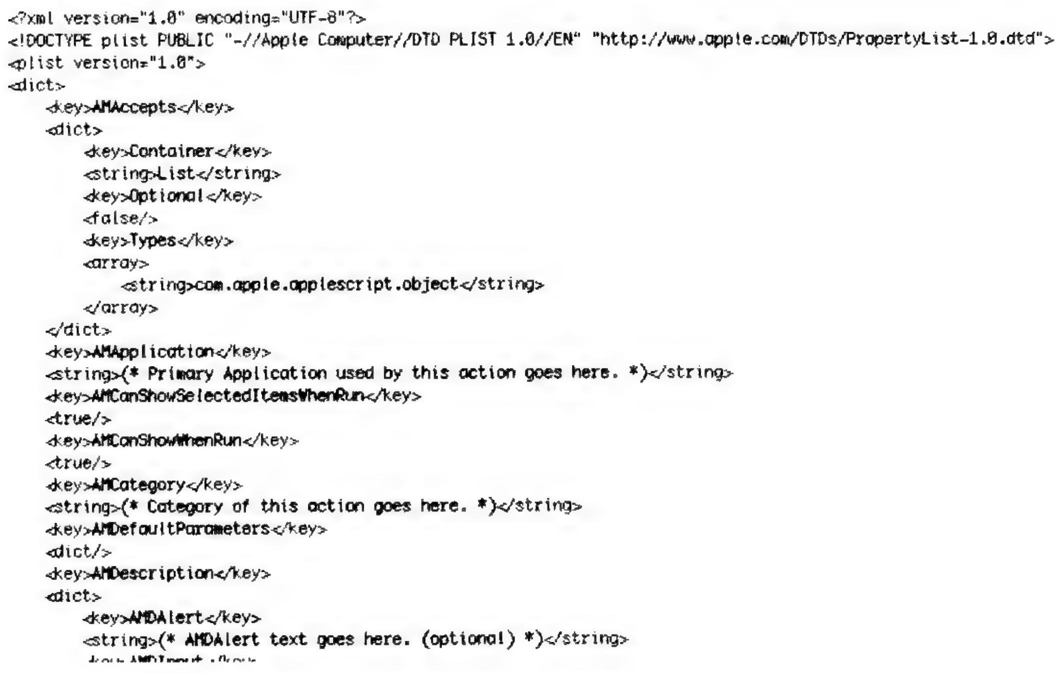
Because the values of some Automator properties appear in the user interface, you should include translations of them in an `Infoplist.strings` file for each action localization you provide. See "Internationalizing the Action" (page 40) for further information.

In addition to properties that are specific to Automator, an action's `Info.plist` file contains properties that are common to all bundles, including applications as well as action bundles. The values of most of these generic bundle properties are supplied automatically when you create a project. However, you must specify the value of the `CFBundleIdentifier` property. Automator uses this property to find an action and its resources. The identifier needs to be unique, and should use the standard format:

`com.`*CompanyName*`.Automator.`*ActionIdentifier*

For example, if your company's name is Acme and your action is named Find Foo Items, a suitable `CFBundleIdentifier` would be `com.Acme.Automator.FindFooItems`.

Figure 6 Part of the template for the Automator properties in Info.plist



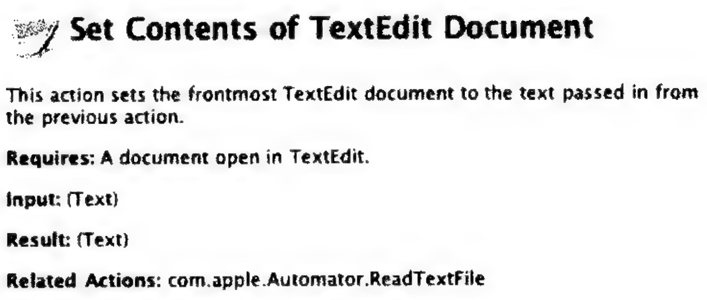
```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple Computer//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>AMAccepts</key>
    <dict>
        <key>Container</key>
        <string>List</string>
        <key>Optional</key>
        <false/>
        <key>Types</key>
        <array>
            <string>com.apple.applescript.object</string>
        </array>
    </dict>
    <key>AMApplication</key>
    <string>(* Primary Application used by this action goes here. *)</string>
    <key>AMCanShowSelectedItemsWhenRun</key>
    <true/>
    <key>AMCanShowWhenRun</key>
    <true/>
    <key>AMCategory</key>
    <string>(* Category of this action goes here. *)</string>
    <key>AMDefaultParameters</key>
    <dict/>
    <key>AMDescription</key>
    <dict>
        <key>AMDAlert</key>
        <string>(* AMDAlert text goes here. (optional) *)</string>
```

You can edit the information property list as a text file in Xcode. If you would like always to open up the `Info.plist` file in a different editor for property lists, such as the Property List Editor application or BBEdit, use the Finder's Get Info window to set the default application for files with extension `.plist`. Then in Xcode, use the contextual menu command Open with Finder.

Writing the Action Description

A small but important part of action development is writing the action description. Automator displays the description it its lower-left view whenever the user selects the action. The description briefly describes what the action does and tells users anything else they should know about the action. Figure 7 shows what a typical description looks like.

Figure 7 An sample action description



Set Contents of TextEdit Document

This action sets the frontmost TextEdit document to the text passed in from the previous action.

Requires: A document open in TextEdit.

Input: (Text)

Result: (Text)

Related Actions: com.apple.Automator.ReadTextFile

Because the description fits into a relatively small area of the Automator window, you should make it as concise and brief as possible. Ideally the user should not have to scroll the description view to see all of the text.

A description has several parts, each of which you specify through an Automator property in the bundle's information property list (`Info.plist`):

- **Icon**. A 32 x 32 pixel TIFF image displayed in the upper-left corner of the description. It is the application icon that the `AMIcon` property specifies (see "Property Keys and Values" (page 69)).
- **Title**. The string that you specify for the `AMName` property (see "Property Keys and Values" (page 69)).
- **Summary**. A sentence or two directly under the title that succinctly states what the action does.
- **Input** and **Result**. States the types of data that the action accepts and provides. Automator enters default values for these sections if you do not specify anything.
- **Options**. Summarizes the configuration options on the action's user interface.
- **Requires**. Describes any requirement for the action to work properly—for example, Safari must be displaying a web page.
- **Alert**. Warns the user of any likely consequence of the action—for example, if it will delete a calendar.
- **Note**. Presents additional information that is not as critical as an alert.
- **Related actions**. Indicates actions that are related to this one—for example, an Import Image action might mention an Export Image action.

A description's icon, title, summary, input, and result sections are required or strongly recommended. All of the properties listed above, except for icon and title, are subproperties of the description-specific `AMDescription` property. The `AMDescription` example in Listing 1 shows the subproperties specified for the Send Birthday Greeting description depicted in Figure 7 (page 38).

Listing 1 AMDescription properties for Send Birthday Greetings description

```
<key>AMDescription</key>
<dict>
    <key>AMDInput</key>
    <string>Address Book entries from previous action.</string>
    <key>AMDOptions</key>
```

```
        <string>Birthday message. Choice of picture.
            Randomly chosen picture. </string>
        <key>AMDSummary</key>
        <string>This action sends an email birthday greeting, with a
            picture, to each of the Address Book entries.</string>
    </dict>
```

See "Property Keys and Values" (page 69) for further information on the `AMDescription` keys and values.

Important: If you are deploying the action in multiple localizations, make sure you create a separate description for each localization. See "Internationalizing the Action" (page 40) for further information.

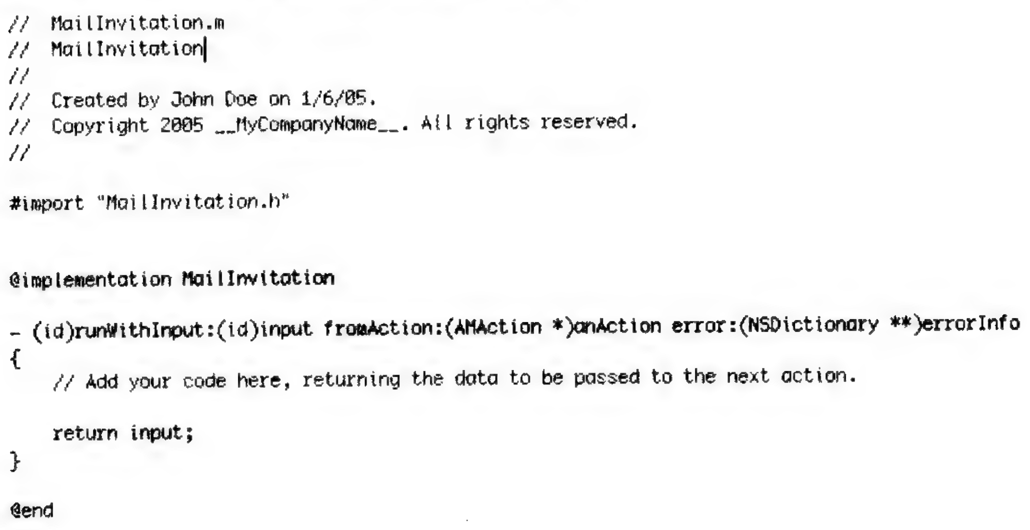
Writing the Action Code

The most important step in creating an action is writing the Objective-C or AppleScript code (or Objective-C *and* AppleScript code) that implements the logic for your action. The project template for Automator actions gives you template files for action implementation:

- `projectName.h` and `projectName.m` files for Objective-C-based actions
- `main.applescript` for AppleScript-based actions

The template files fill out as much of the required structure as possible. The Objective-C header file, for example, has the necessary framework imports and an `@interface` declaration showing inheritance from AMBundleAction. The Objective-C implementation includes a null implementation of the method that all actions must implement, `runWithInput:fromAction:error:` (see Figure 8). The `main.applescript` file, on the other hand, has a skeletal structure for the `on run` command handler that all AppleScript-based actions must implement.

Figure 8 Template for an Objective-C action implementation



```
//  MailInvitation.m
//  MailInvitation
//
//  Created by John Doe on 1/6/05.
//  Copyright 2005 __MyCompanyName__. All rights reserved.
//

#import "MailInvitation.h"


@implementation MailInvitation

- (id)runWithInput:(id)input fromAction:(AMAction *)anAction error:(NSDictionary **)errorInfo
{
    // Add your code here, returning the data to be passed to the next action.

    return input;
}

@end
```

See "Implementing a Script-Based Action" (page 53) and "Implementing an Objective-C Action" (page 59) for requirements, suggestions, and examples related to implementing an Automator action.

Internationalizing the Action

Most polished applications that are brought to market feature multiple localizations. These applications not only include the localizations—that is, translations and other locale-specific modifications of text, images, and other resources—but have been internationalized to make those localizations immediately accessible. Internationalizing involves a number of techniques that depend on a "preferred languages" user preference and a system architecture for accessing resources in bundles. Loadable bundles (which, of course, are bundles just as applications are) depend upon the same mechanisms for internationalization.

The following sections summarize what you must do to internationalize your actions. For the complete picture, see *Internationalizing Your Software* for a general description of internationalization and *Internationalization*, for a discussion that is specific to Cocoa.

Localized Strings

If your action programmatically generates strings and you want localized versions displayed, you need to internationalize your software using access techniques that are different for Objective-C actions and script-based actions.

Note: Programmatically generated strings have their basis in code, such as a message that appears in an error dialog. They do not include text that appears in a user interface unarchived from a nib file. Nib files are localized just as are other resource files; see "Internationalizing Resource Files" (page 42) for a summary of the procedure.

Both Objective-C and AppleScript actions require you to create for each localization a strings file, which is a file with an extension of `.strings`. (The conventional, or default, name for a strings file is `Localizable.strings`.) Each entry in a strings file contains a key and a value; the key is the string in the development language and the value is the translated string. An entry can also have a comment to aid translators. Use a semicolon to terminate an entry. Here are a few examples:

```
/* Title of alert panel which brings up a warning about
saving over the same document */
"Are you sure you want to overwrite the document?" =
"Souhaitez-vous vraiment écraser le document ?";

/* Encoding pop-up entry indicating automatic choice of encoding */
"Automatic" = "Automatique";

/* Button choice allowing user to cancel. */
"Cancel" = "Annuler";
```

When you have completed a `Localizable.strings` file for a localization, you must internationalize it just as you would any other language- or locale-specific resource file of the bundle. See "Internationalizing Resource Files" (page 42) for a summary.

For Objective-C code, use the `NSLocalizedString` macro or one of the other `NSLocalizedString...` macros to request a localization appropriate for the current user preference. Listing 2 gives an example that shows the use of `NSLocalizedString` in conjunction with the NSString `stringWithFormat:` method.

Listing 2 Using NSLocalizedString in Objective-C code

```
- (NSString *)displayName
{
    int cnt = [pdfView pageCount];
    NSString *name;

    if (cnt == 1) {
        NSString *format = NSLocalizedString(@"%@ (1 page)",
            @"Window title for a document with a single page");
        name = [NSString stringWithFormat:format, fileName];
    } else {
        NSString *format = NSLocalizedString(@"%@ (%d pages)",
            @"Window title for a document with multiple pages");
        name = [NSString stringWithFormat:format, fileName, cnt];
    }

    return name;
}
```

For AppleScript scripts the command equivalent to `NSLocalizedString` is `localized string`. One good approach is to have a local subroutine that takes the string to localize as a parameter and uses the `localized string` command on it, as in Listing 3.

Listing 3 Script handler for localizing strings

```
on localized_string(key_string)    return localized string key_string in bundle
 with identifier "com.apple.Automator.myAction"end localized_string
```

Elsewhere in `main.applescript` and in other scripts for the action, call this subroutine when you need to get a string in the current localization:

```
    if the calendar_name is my localized_string("No Calendars") then
error my localized_string("The copy of iCal on this computer contains no calendars
 to clear.")
```

Localizing the Automator Properties

A standard variant of a strings file for projects is `Infoplist.strings`. In this file you assign localized strings (that is, translations) to the keys that appear in the `Info.plist` file. For Automator, this includes not only the top-level properties such as `AMName` but subproperties of Automator properties. For example, the following excerpt is from the `Infoplist.strings` file for the Crop Images action:

```
AMName = "Crop Images";
ApplyButton = "Add";
IgnoreButton = "Don't Add";
Message = "This action will change the image files passed into it.  Would you
like to add a Copy Files action so that the copies are changed and your originals
 are preserved?";
```

The key-value pairs in this example include not only an English-localized value for the `AMName` property but localized strings for the subproperties of the `AMWarning` property.

Internationalizing Resource Files

You should internationalize any file in your action project that contains data that is specific to a language or locale. These files include:

- Nib files
- Image files
- Strings files (`Localizable.strings`)
- `Infoplist.strings`

Internationalization of a resource file simply entails the assignment of a language or locale to the file. When you build the action project, Xcode automatically puts the resource in a localization subdirectory of the bundle's `Resources` directory. Localization subdirectories have an extension of `.lproj` (for example, `French.lproj`).

To internationalize a resource file, do the following:

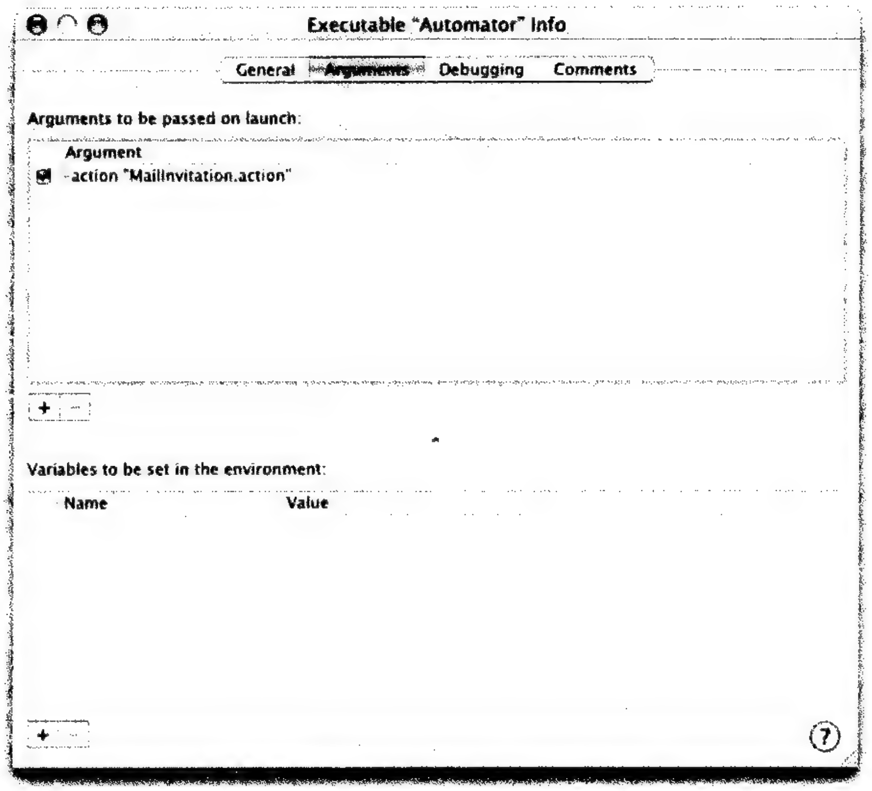
1. Select the Resources folder in Xcode.
2. Add the file to the project (Project > Add Files).
3. Select the file and choose Get Info from the Project menu.

4. Click Make File Localizable.
5. Click Add Localization and enter or select a language or locale.

Testing, Debugging, and Installing the Action

An action project is automatically set up in Xcode so that when you choose the Run or Debug commands, Xcode launches the Automator action and adds your action to the set of actions loaded by Automator. To see how this is done, select Automator in the Executables smart group and choose Edit Active Executable from the Project menu. As you can see, in the General pane of the inspector the "Executable path" value is set to `/Applications/Automator.app`. Select the Arguments pane of the inspector and note that the `-action` launch-time argument has been set to your action (Figure 9).

Figure 9 Setting the launch argument for Automator


Executable "Automator" Info
General
Arguments
Debugging
Comments
Arguments to be passed on launch:
Argument
+
-
Variables to be set in the environment:
Name
Value
+
-
?

-action "MailInvitation.action"

Testing and Debugging Stragegies

To test your action, build and run the action project (Build > Build and Run). When Automator launches, create a workflow in which your action might appear. Run the workflow and observe how your action performs. To get better data from testing, consider the following steps:

- Use the View Results action to see the output of prior actions.
- Use the Confirmation Dialog action to pause or cancel execution of the workflow.
- Add one each of the above two actions between each action.

If the action is based on AppleScript, you can use the Run AppleScript action to test your `on run` command handler as you write it.

If your action is based on an Objective-C implementation, you can debug an action just as you would any other Objective-C binary. Simply set a breakpoint in Xcode. When your action is run in Automator, the breakpoint will trigger in `gdb`. To debug AppleScript actions, insert `log` or `display dialog` statements in the code.

Installing Actions

When your action has been thoroughly debugged and tested, build a deployment version of the bundle (using the proper optimizations). Then create an installation package for the action (or add the action to your application's installation package). The installer should copy the action to `/Library/Automator` or `~Library/Automator`, depending on whether access to the action should be system wide or restricted to the installing user.

Instead of installing your action separately, you can put it inside the bundle of your application, especially if the action uses the features of that application. When Automator searches for actions to display, it looks inside registered applications as well in the standard Automator directories. The advantage of packaging your actions inside an application is that you don't need to create a separate installation package to install the actions. To install the actions, users need only drag the application to a standard location.

Action bundles should be stored inside the `Contents` directory of the application wrapper at `/Library/Automator`. Thus, if your action is `MyAction.action` and your application is `MyApp.app`, the path inside the application would be:

```
MyApp.app/Contents/Library/Automator/MyAction.action
```

You can either manually copy an action into this location (after creating the necessary subdirectories) or you have Xcode copy it using a Copy Files build phase. If you copy an action into an application bundle but the application is already installed on a system, you must get Launch Services to recognize that the application has new content to register (that is, the new action) by changing the application's modification date. You can do this by entering the `touch` command in the Terminal application.

```
$> sudo touch /Applications/MyApp.app
```

Or you can rename the application in Finder to something else, change it back to the original name, and then launch the application once.

Show When Run

Automator has two categories of users. The first is a user who creates a workflow by putting actions in a sequence that accomplishes a certain goal. This type of user can then save the workflow as an application which another user—a user of the second kind—can then double-click to run. If the workflow creator is the only user who can set the parameters of workflow actions (by choosing items and entering values in action views), users in the second category are at a disadvantage. They are compelled to accept the choices of the workflow creator.

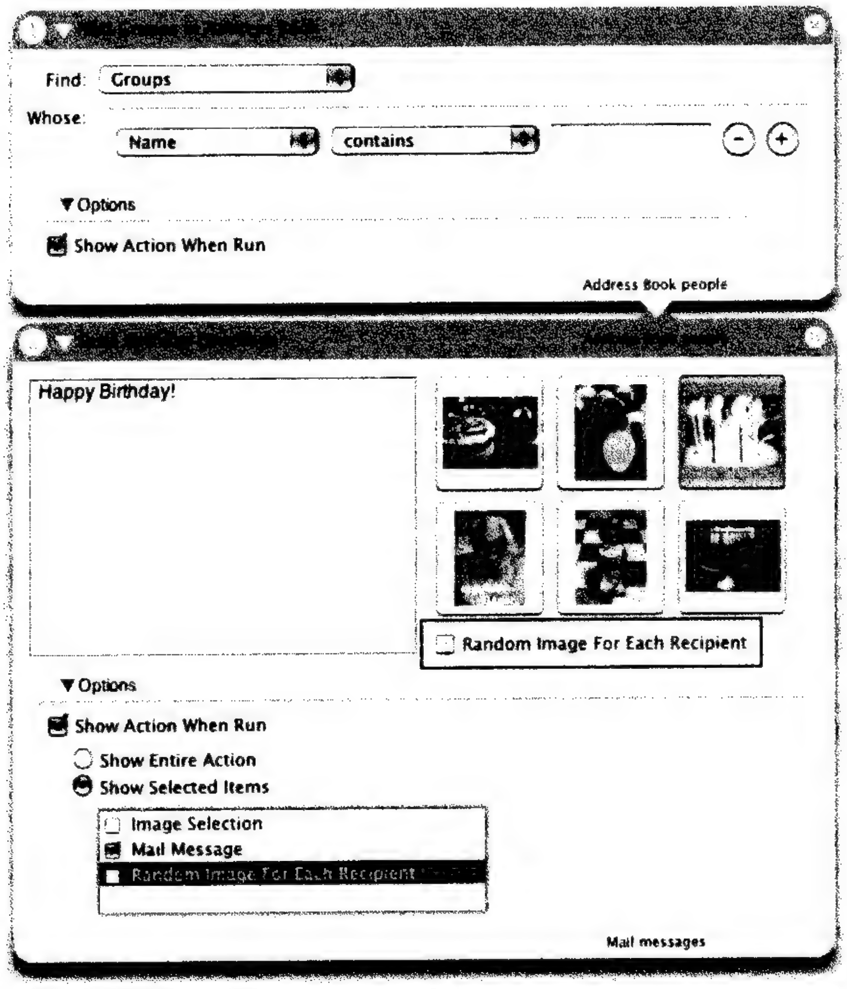
To get around this problem, Automator has the Show When Run feature. If this feature is turned on for an action in a workflow, when Automator executes the workflow it displays the user interface (in whole or part) for the action when execution reaches that point. The workflow user—as opposed to the workflow creator—can then make the required settings before the action proceeds.

The following sections describe Show When Run in more detail, show what it looks like, and explain how developers can modify the default behavior.

The "Show Action When Run" Option

The views of most actions in a workflow include a triangular disclosure button in their lower-left corner labeled "Options". When a workflow creator clicks this control, the bottom of the view expands to expose a new set of controls. The number of controls in this disclosed view vary according to how the action view is configured. Figure 1 illustrates the two kinds of control sets.

Figure 1 The Show Action When Run option in two actions



☐ Random Image For Each Recipient

✓ Show Action When Run (first action)

✓ Show Action When Run (second action)

○ Show Entire Action

● Show Selected Items

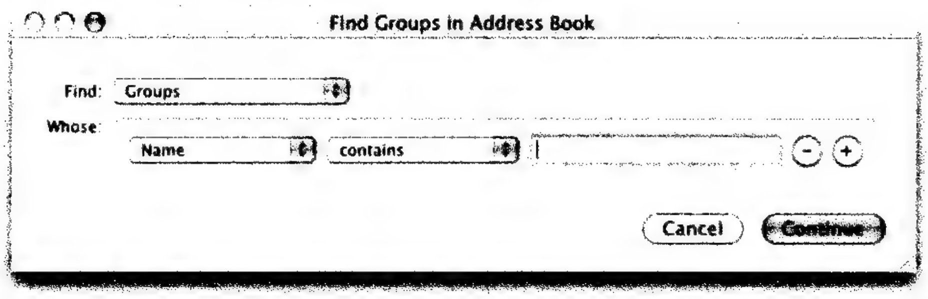
☐ Image Selection

✓ Mail Message

☐ Random Image For Each Recipient

In both exposed Option subviews, the first control is a check box labeled "Show Action When Run." However, in the first action that check box is also the only control. If the check box is clicked in the first Option subview and the workflow is run, the action displays its entire user interface in a separate window, as shown in Figure 2.

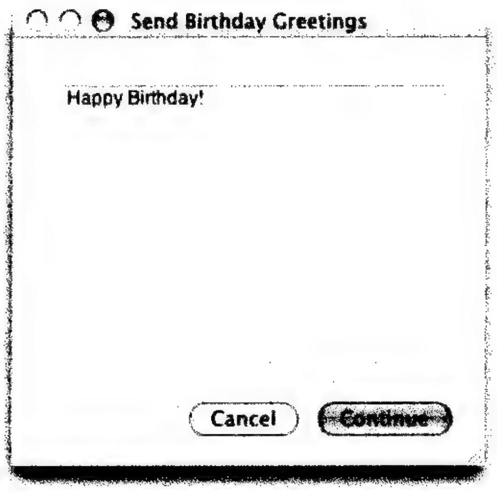
Figure 2 Action displaying entire user interface in a window



The user makes selections and fills in information in this window and clicks Continue to have the action proceed.

In the second Option subview (as shown in Figure 1 (page 46)), when the user selected "Show Action When Run", the two radio buttons under it were enabled. He or she then clicked the "Show Selected Items" button and selected the "Mail Message" check box in the table. As shown in Figure 3, when Automator reaches this action in the workflow, it displays this portion of the action view in a separate window.

Figure 3 Action displaying part of its user interface in a window



In summary, an action may present one of three styles of the Show When Run feature:

- No Options disclosure button is presented. This Show When Run style is for actions where only the workflow creator can decide what the action parameters should be.
- The action displays the Options disclosure button and, when that is clicked, only the Show Action When Run check box. This style is used when Automator should present all controls of the action to users when the workflow executes.

- The action displays the Options disclosure and, when that is clicked, the "Show Entire Action" and "Show Selected Items" radio button along with the selected-items table view. This style of Show When Run is appropriate for actions where workflow users should be able to make only some action settings. This style is the default configuration,

Refining Show When Run

In the default Show When Run configuration, Automator displays the Options disclosure button and allows workflow creators to select one or more parts of the action view for Automator to display when the action is run in a workflow. Developers can refine the default appearance and behavior of the Show When Run feature in two ways: by specifying new values for certain Automator properties and by grouping controls in special boxes in Interface Builder.

Specifying Properties for Show When Run

These three general styles of the Show When Run feature discussed in "The "Show Action When Run" Option" (page 45) are controlled by two Automator properties:

- `AMCanShowWhenRun` with a value of `<true/>` tells Automator to display at least the Options disclosure button and the "Show Action When Run" check box for the action. If the value is `<false/>`, the button does not appear in the action view.
- `AMCanShowSelectedItemsWhenRun` with a value of `<true/>` tells Automator to include the radio buttons "Show Entire Action" and "Show Selected Items" radio buttons along with the table view of control subsets. If the value is `<false/>` only the "Show Action When Run" check box is shown.

  This property has no effect if `AMCanShowWhenRun` is `<false/>`.

The default Show When Run configuration for actions is to have both properties turned on (that is, Automator assumes a value of `<true/>`.) In this style, the Option button is displayed, and when that button is clicked all "Show Action When Run" options are presented, including the table allowing workflow writers to select groups of controls to display to end users.

If you want only the "Show Action When Run" check box to appear in the exposed view under Options, specify `<false/>` for `AMCanShowSelectedItemsWhenRun`. If you don't want the Options disclosure button to appear at all, specify `<false/>` for `AMCanShowWhenRun`.

Grouping User-Interface Objects

By default, Automator populates the selected-items table with check box items using the following algorithm:

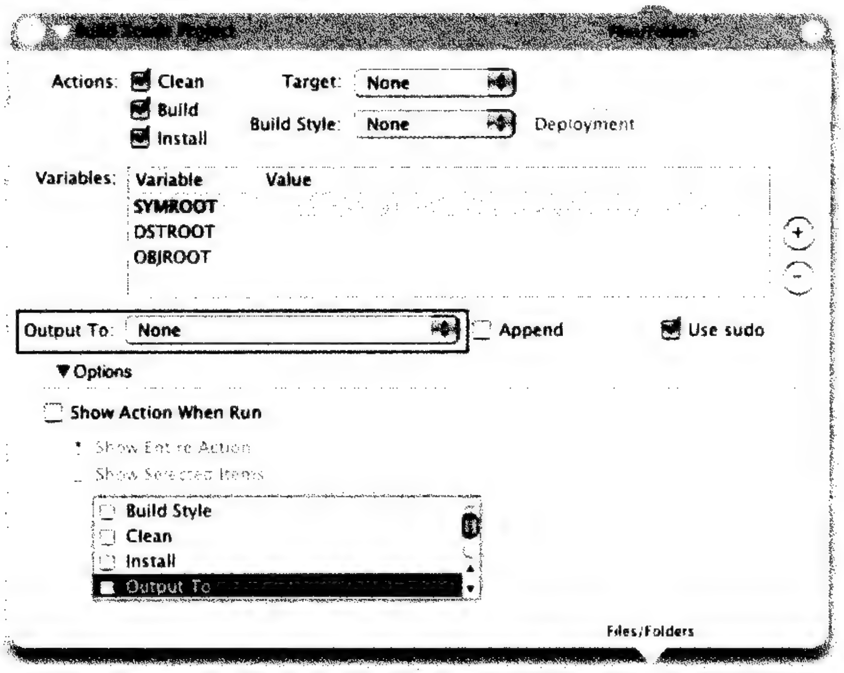
1. It goes through the items in the top level of the action view looking for eligible objects (such as pop-up lists, text fields, and buttons).
2. When it finds an eligible object, it determines if there is a label to the left of the object; it there is, it uses the label in the selected-items table view.

3. If it cannot find an associated label it uses an appropriate name such as "Text field 1" or "Item 3". (The bottom action displayed in Figure 1 (page 46) includes an "Item 1" check box in its table.)

These semi-arbitrary names that Automator assigns to some items in the table are often confusing to users. What does "Text field 1" refer to, especially if there is more than one text field? Moreover, it sometimes makes more sense to group related controls under one item. For example, if selection of an item in a pop-up list results in a text field being enabled, the text field and the menu item belong together.

Interface Builder has been enhanced to give developers of Automator actions the ability to group individual objects on action views under a given name. When you have completed the procedure in Interface Builder, the Options subview presented to workflow creators will look more like the example in Figure 4, which contains no items with names such as "Item 1".

Figure 4 An action with assigned groups in the selected-items table



Actions: ☑ Clean ☑ Build ☑ Install

Target: None

Build Style: None Deployment

Variables:

Variable	Value
SYMROOT	
DSTROOT	
OBJROOT	

Output To: None ☐ Append ☑ Use sudo

▼ Options

☐ Show Action When Run

Show Entire Action

Show Selected Items

☐ Build Style
☐ Clean
☐ Install
☐ Output To

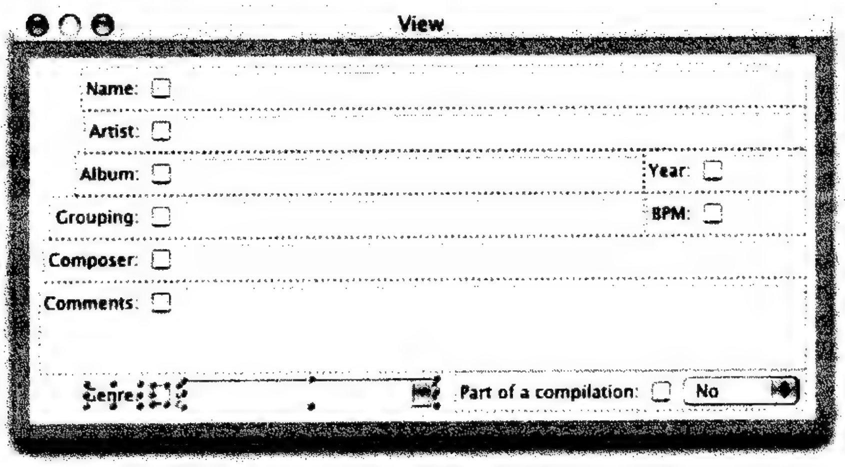
Files/Folders

To make named groups of action-view objects, complete the following procedure with the `main.nib` file open in Interface Builder:

1. Make sure that the Automator palette is loaded. If the palette window does not have a palette named Cocoa-Automator, choose Tools > Palettes > Palette Preferences and load `AMPalette.palette`.

2. Select the objects you want in the group by Shift-clicking each object in turn (Figure 5).

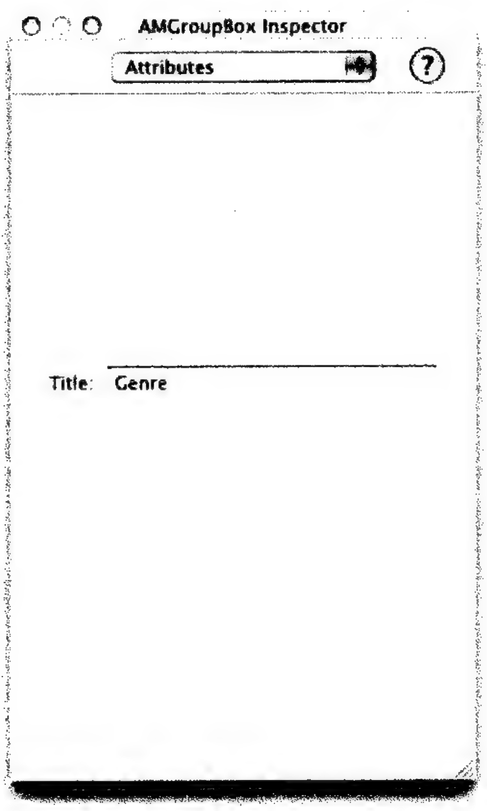
Figure 5 Selecting objects for an action group



3. Choose Layout > Make subview of > Automator Box. The grouped objects are enclosed by a colored rectangle.

4. In the Attributes inspector (Command-1) enter a title for the group (Figure 6). This title is what appears in the selected-items table to identify the group.

Figure 6 Assigning a title to the action group



Title: Genre

The box that groups a set of controls in an action view is an instance of AMGroupBox, a subclass of NSBox created specifically for actions. Developers whose actions are based on AppleScript should keep in mind that that the AMGroupBox instance changes the view hierarchy of the objects in an action view. However, because AMGroupBox is a subclass of NSBox, the "box" terminology stilll works in scripts.

Implementing a Script-Based Action

The Xcode template for an AppleScript Action project includes an uncompiled script file named `main.applescript`. An AppleScript-based action must, at the minimum, add to `main.applescript` the scripting code that accesses the services of one or more applications and completes the stated purpose of the action. An AppleScript-based action can also rely on other components for its implementation, such as:

- Other AppleScript script files, particularly for handling user-interface events
- Objective-C classes, particularly for accessing system resources
- Command-line tools or shell scripts

Regardless of the possible implementation components, an AppleScript-based action always has a `main.applescript` file and an invisible instance of AMAppleScriptAction. This instance owns the action bundle's nib file, has access to `main.applescript`, and provides a default implementation of the method used to run the action behind the scenes, `runWithInput:fromAction:error:`. (AMBundleAction, the superclass of AMAppleScriptAction, declares this method.)

The Structure of the on run Command Handler

The template `main.applscript` contains only the following skeletal structure of the `on run` command handler:

```
on run {input, parameters}
    -- your code goes here
    return input
end run
```

This `on run` command handler has two positional parameters, `input`, and `parameters`, The `input` parameter contains (in most instances) the output of the previous action in the workflow; it is almost always in the form of a list. The `parameters` parameter contains the settings made in the action's user interface; the values in `parameters` (a record object) are typically set via the Cocoa bindings mechanism but can also be manually updated. The template code finally returns `input` as its output; your action should always return something as output, even if it what is given it as input.

You can use any valid AppleScript command, object, reference, expression, and statement in `main.applescript`. However, the following steps are suggested for an action's `run` handler (not all steps are applicable to every kind of action).

- Set local variables to the values of the current parameters. To extract the values from the `parameters` parameter, use the keys specified for bindings. For example:

  ```
  set multiple_selection to |multipleSelection| of parameters
  set prompt_message to |promptMessage| of parameters
  set default_location to |defaultLocation| of parameters
  ```

  You use these parameters to control the logic of the script.
- Set a local variable for output to an empty list. For example:

  ```
  set output to {}
  ```

- In a `repeat` loop, perform whatever transformations are necessary on the items in the input list and add each transformed item to the output list. For example:

  ```
  repeat with i in input
      copy (POSIX path of i) to end of filesToCopy
  end repeat
  ```

- As the last step, return the output list.

Some actions in their `AMAccepts` property specify that their input is optional (that is, the `Optional` subproperty is set to `<true/>`). This setting allow users to select a "Ignore Result From Previous Action" pop-up item in the action's view. If your action has made input optional in its `AMAccepts` property, you need to determine if the user has made this choice by testing the `ignoresInput` field of `parameters`; if it is set to `true`. then you do not need to do anything with the input object.

You can include subroutines in `main.applescript`. For example, you might want to write a subroutine for returning localized strings (described in "Localized Strings" (page 40)).

You can execute command-line tools or shell scripts for your script. The following script fragment runs the `screencapture` tool:

```
set screencapture to "/usr/sbin/screencapture "
-- here set up arguments of screencapture
do shell script screencapture
```

A caveat about using the `do shell script` command is that, because of the threading architecture, it makes the user interface unresponsive until the shell script completes or th e user presses Command-. (period).

An action's scripts can also invoke Objective-C methods that you have implemented in a custom class; for example, the following statement calls a class method named `isDVDPlayerRunning`:

```
set dvdRunning to call method "isDVDPlayerRunning" of class "TakeScreenshot"
```

Note: The `call method` command is documented in *AppleScript Studio Terminology Reference*.

"Hybrid Actions" (page 56) discusses actions that contain a mix of AppleScript and Objective-C code.

Other AppleScript Scripts

An AppleScript action can contain scripts other than `main.applescript`. These additional scripts typically contain two types of handlers:

- AppleScript Studio-specified event handlers for managing the user interface of the action. For example, in an event handler you can modify the presentation of the user interface based on the user's choices.
- Handlers for the Automator-defined commands `update parameters` and `parameters updated`, which are attached to an action's view.These handlers are an alternative to Cocoa bindings: you can use them to store user settings made in the action parameters and to refresh those settings from existing parameters. For more on these handlers, see "Updating Non-bound Parameters" (page 57).
- Handlers for the Automator-defined commands `activated` and `opened`, which are attached to an action's view. The `on activated` handler is called when the action's workflow is activated, and the `on opened` is called when the user adds the action to the workflow. You should use the `on opened` handler when you need to populate your action with data that takes some time to get and consequently need to display a progress indicator.

AppleScript Studio allows you to request handlers for the `update parameters`,`parameters updated`, `activated`, and `opened` commands in the Automator Action View group in the AppleScript pane of the Interface Builder inspector.

As an example of AppleScript Studio event handlers, consider a pop-up list for various date formats. When the user chooses a item from this pop-up list, the action changes an example showing the effect of this format change. Listing 1 shows an event handler that is invoked when the user chooses a pop-up item for saving a screen capture to a "Specific File".

Listing 1 An event handler for displaying a Save panel

```
on clicked theObject
    if name of theObject is "choose output" then
        -- Setup the properties in the 'save panel'
        tell save panel
            set title to "Choose an output file to save the screenshot to"
            set prompt to "Save"
            set required file type to "pdf"
            set treat packages as directories to false
        end tell

        if (display save panel in directory "" with file name "") is equal to 1 then
            set contents of text field "output file" of super view of theObject to path
                name of save panel
            set contents of popup button "output choice" of super view of theObject to 1
            set enabled of button "choose output" of super view of theObject to true
            set enabled of text field "output file" of super view of theObject to true
        end if
    end if
end clicked
```

Hybrid Actions

An action implementation can contain a mix of AppleScript and Objective-C code. Usually such hybrid actions are AppleScript-based—in other words, they have a `main.applescript` file containing the `run` command handler. However, it is possible to have Objective-C actions that use the Cocoa scripting classes to load (or create) and execute scripts. See "Implementing an Objective-C Action" (page 59) for a discussion of the latter type of hybrid action.

Why would you want to create a hybrid action? One reason is to have greater control of the user interface. An Objective-C class that links with the Application Kit has access to the full range of programmatic resources for managing the appearance and behavior of user-interface objects.

But perhaps the most common reason for hybrid actions is to get access to system resources that a script could not get on its own. Scripts can call Objective-C instance methods with the `call method` command, but only if the target object is something that the script can address (for example, `call method "title" of window 1`). Sometimes, an AppleScript Studio script does not have access to a valid object for a `call method` command. To get around this limitation, you can create a simple custom class that implements one or more class methods. The first argument of these methods is the target (receiver) of an instance method that the class method wraps. Listing 2 illustrates this approach.

Listing 2 Implementing a custom Objective-C method to be called by a script

```
#import "CreatePackageAction.h"

@implementation CreatePackageAction

+ (BOOL)writeDictionary:(NSDictionary *)dictionary withName:(NSString *)name
{
    return [dictionary writeToFile:name atomically:YES];
}

@end
```

In a script you can then use a `call method "m" of class "c" with parameters{y,z}` statement to call this method (Listing 3).

Listing 3 A script calling the Objective-C method

```
set descriptionRecord to
        {|IFPkgDescriptionTitle|packageTitle,|IFPkgDescriptionVersion|:packageVersion,
        |IFPkgDescriptionDescription|:description,
        |IFPkgDescriptionDeleteWarning|:deleteWarning}
set rootName to call method "lastPathComponent" of rootFilePath
set descriptionFilePath to temporaryItemsPath & rootName & "description.plist"
call method "writeDictionary:withName:" of class "CreatePackageAction" with parameters
     {descriptionRecord, descriptionFilePath}
```

Updating Non-bound Parameters

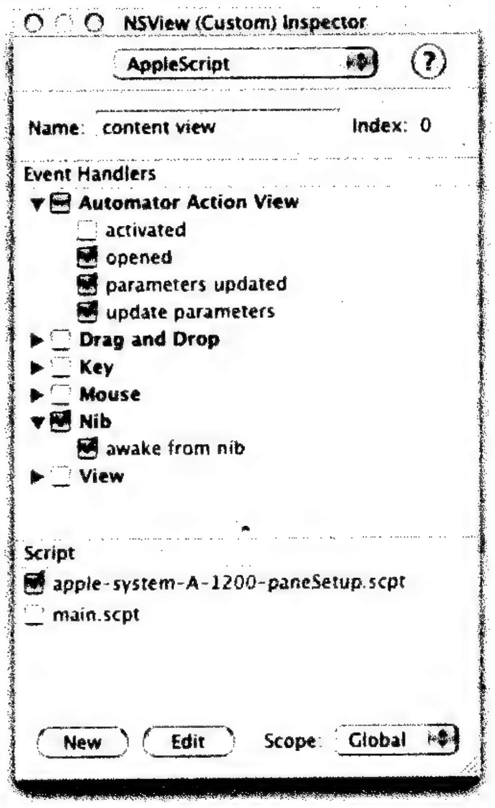
A property of all actions, including AppleScript-based ones, is a record (equivalent in Objective-C to a dictionary) containing values reflecting the settings users have made in the action's user interface. When Automator runs an AppleScript action by calling its `run` command handler, it passes in these values in the `parameters` parameter. Most actions use the Cocoa bindings mechanism to set the parameters property. However, you may choose to forgo bindings in favor of more direct approach.

The Automator application defines several commands for AppleScript Studio. Among these are `update parameters` and `parameters updated`, which can be attached to an action's view. The `update parameters` command is sent when the an action's parameters record need to be refreshed from the values on the user interface. The `parameters updated` command is sent when the parameters record changes any of its values.

To specify handlers for these commands in Interface Builder, follow the usual AppleScript Studio procedure:

1. Select the action's view.
2. In the AppleScript pane of the Info window, expose the commands under Automator Action View (see Figure 1).
3. Click the check boxes for `update parameters` and `parameters updated`.
4. Specify a script file for the command handlers.

Figure 1 Selecting the parameters-related handlers for action views


NSView (Custom) Inspector

AppleScript

Name: content view Index: 0

Event Handlers
- [✓] Automator Action View
  - [ ] activated
  - [✓] opened
  - [✓] parameters updated
  - [✓] update parameters
- [ ] Drag and Drop
- [ ] Key
- [ ] Mouse
- [✓] Nib
  - [✓] awake from nib
- [ ] View

Script
- [✓] apple-system-A-1200-paneSetup.scpt
- [ ] main.scpt

New Edit Scope: Global

Automator calls the handler for `update parameters` just before it runs the action. In this handler, the action should get the settings from the user interface and update the parameters record with them. Listing 4 provides an example of an `update parameters` handler.

Listing 4 An "update parameters" command handler

```
on update parameters theObject parameters theParameters
    set |destinationPath| of theParameters to (the content of text field "folder path" of
 the destination_folder_box) as string
    set |textInput| of theParameters to (the content of text field "text input" of the
text_source_box) as string
    set |fileName| of theParameters to (the content of text field "file name" of the
file_name_box) as string
    set |chosenVoice| of theParameters to the title of popup button "voice popup" of
voice_box
    set |textSource| of theParameters to ((the current row of matrix "source matrix" of
the text_source_box) as integer) - 1
    return theParameters
end update parameters
```

Always return the updated parameters object as the final step in this handler (`theParameters` in the above example).

Implementing an Objective-C Action

The particular advantage of actions written in Objective-C (compared to AppleScript actions) is that they have greater access to the range of system resources of Mac OS X. Actions that use Objective-C code can incorporate the features not only of Objective-C frameworks such as Foundation and Application Kit but of virtually any other framework (Because Objective-C is a superset of ANSI C, an Objective-C class can call functions published in a C interface). Purely Objective-C actions have an offsetting disadvantage: they cannot easily control applications or access their features except for those applications that offer a public API.

In general, data that Automator pipes through the actions of a workflow is assumed to be AppleScript-compatible. If it detects that an action is based on Objective-C, however, Automator converts it into an Objective-C object that the action can deal with. It also takes the objects that Objective-C actions provide and converts them into a form suitable for AppleScript-based actions.

Specifying Cocoa Type Identifiers

An action that is purely based on Objective-C must have Cocoa-specific type identifiers for its `AMAccepts` and `AMProvides` properties. Currently there are three Cocoa public type identifiers:

- `com.apple.cocoa.string` —An NSString for text
- `com.apple.cocoa.path` — An NSString for full file-system paths
- `com.apple.cocoa.url` —An NSURL for URLs

For example, if you were to specify `com.apple.cocoa.path` as the sole type identifier of the `AMAccepts` property, the input object in the `runWithInput:fromAction:error:` method would an NSString object representing a path (or an array of such string objects). If you were to specify `com.apple.cocoa.url` as the type identifier of the `AMProvides` property, you would have to return an NSURL object (or an array of NSRUL objects) in your implementation of `runWithInput:fromAction:error:`.

For further discussion of the Cocoa type identifiers, see "Type Identifiers" (page 75).

Note: UTI type identifiers designating other types of input and output data for Objective-C actions are planned for future versions of Mac OS X.

Implementing runWithInput:fromAction:error:

In your custom subclass of AMBundleAction you must override the method `runWithInput:fromAction:error:` (inherited from the AMAction class). Aside from specifying the Cocoa type identifiers in your `AMAccepts` and `AMProvides` properties, this method implementation is the only requirement for a purely Objective-C action.

If an action does not have to deal with the input data handed it—for example, its role is to select some items in the file system—the implementation of `runWithInput:fromAction:error:` can return whatever it is designed to provide without touching the input data. In the example in Listing 1, the action returns a list of paths to movies selected in its user interface (and stored in its parameters):

Listing 1 An Objective-C action that does not handle its input object

```
- (id)runWithInput:(id)input fromAction:(AMAction *)anAction
        error:(NSDictionary **)errorInfo {
    return [[self parameters] objectForKey:@"movieFiles"];
}
```

However, most actions operate on the input data given them from the previous action. The input object and the output object for an action are almost always NSArray objects because the `Container` subproperties of `AMAccepts` and `AMProvides` are by default lists (equivalent to NSArray objects in Objective-C). Many Objective-C actions implement `runWithInput:fromAction:error:` using a general approach that is similar to a typical `on run` handler in an AppleScript-based action:

1. If the container type of the action's `AMProvides` property is List, prepare an output array for later use (usually by creating an NSMutableArray object).

2. Iterate through the elements of the input array and for each perform whatever operation is required and write the resulting data item to the output array.

3. Return the output array.

Listing 2 Implementing runWithInput:fromAction:error:

```
- (id)runWithInput:(id)input fromAction:(AMAction *)anAction error:(NSDictionary
 **)errorInfo
{
    NSMutableArray *returnArray = [NSMutableArray arrayWithCapacity:[input
count]];
    NSEnumerator *enumerate = [input objectEnumerator];
    NSString *xmlFile;

    while (xmlFile = [enumerate nextObject]) {
        NSError *anError=nil;
        NSString *returnStr;
        NSXMLDocument *xmlDoc = [[NSXMLDocument alloc]
initWithContentsOfURL:[NSURL fileURLWithPath:xmlFile] options:nil error:&anError];
        NSLog(@"XML document = %@", [xmlDoc description]);
        if (!xmlDoc) {
            [returnArray addObject:[NSString stringWithFormat: @"ERROR: Could
not make document from file: %@\n", xmlFile]];
            continue;
        }
        if (anError) {
            [returnArray addObject:[NSString stringWithFormat: @"ERROR: Error
in processing file %@, because = %@\n", xmlFile, [anError localizedDescription]]];
            continue;
        }

        NSString *queryStr;
       if ([[[self parameters] objectForKey:@"elementAttributeChoice"] intValue])
 { // attribute
            queryStr = [NSString stringWithFormat:@".//@%@/text()", [[self
parameters] objectForKey:@"elementAttributeName"]];
        } else {
            queryStr = [NSString stringWithFormat:@".//%@/text()",  [[self
parameters] objectForKey:@"elementAttributeName"]];
        }
        anError = nil;
        NSArray *results = [xmlDoc nodesForXPath:queryStr error:&anError];
        if (anError) {
            [returnArray addObject:[NSString stringWithFormat: @"ERROR: Error
in XQuery because = %@\n", [anError localizedDescription]]];
            continue;
        }
        [returnArray addObject:[NSString stringWithFormat:@"\n--------- Found
in file %@ ----------\n", xmlFile]];
        int i, count = [results count];
        for (i = 0; i < count; i++) {
            [returnArray addObject:[results objectAtIndex:i]];
        }
        [xmlDoc release];
    }

    return returnArray;
}
```

Some actions in their `AMAccepts` property specify that their input is optional (that is, the `Optional` subproperty is set to `<true/>`). This setting allow users to select a "Ignore Result From Previous Action" pop-up item in the action's view. If your action has made input optional in its `AMAccepts` property, you need to determine if the user has made this choice by sending the `ignoresInput` method to the previous action in the workflow (the `anAction` parameter); if the response is `YES`, you do not need to do anything with the input object.

If your action encounters an error that prevents it from proceeding, it should give information describing the error to Automator, which then stops executing the workflow and displays an error alert. The last parameter of the `runWithInput:fromAction:error:` method is a pointer to an NSDictionary object. To report errors, you must create a dictionary containing two key-value pairs and return this object to Automator indirectly. The two dictionary properties are:

- `OSAScriptErrorNumber`—Takes an error number encapsulated as an NSNumber object.

For suitable error codes, see the `MacErrors.h` header file of the Carbon Core framework, especially the error codes specific to the OSA framework.

- `OSAScriptErrorMessage`—Takes an NSString object containing a description of the errror.

Because the code example in Listing 2 (page 60) output is text, it reports errors such as the lack of a valid document as part of the output. But what if this were a fatal error that required the action to stop executing? Listing 3 illustrates how you might handle such an error.

Listing 3 Reporting an fatal error to Automator

```
if (!xmlDoc) {
    NSArray *objsArray = [NSArray arrayWithObjects:
        [NSNumber numberWithInt:errOSASystemError],
        [NSString stringWithFormat:@"ERROR:
            Could not make document from file: %@\n", xmlFile], nil];
    NSArray *keysArray = [NSArray arrayWithObjects:OSAScriptErrorNumber,
        OSAScriptErrorMessage, nil];
    errorInfo = [NSDictionary dictionaryWithObjects:objsArray
        forKeys:keysArray];
    return nil;
}
```

Your action has the entire gamut of possibility available to Cocoa applications. It can message Cocoa-framework objects and custom objects from its AMBundleAction subclass, and it can define and implement as many other supporting classes, protocols, and categories as is needed to accomplish the task of the action.An Objective-C action could choose to do any of the following things:

- Execute a command-line tool using an NSTask object.
- Perform animation using NSAnimation or NSViewAnimation.
- Spin off multiple secondary threads using Foundation's threading support.
- Communicate with other applications using distributed objects or distributed notifications.
- Use a C-language framework; for example, you could use Core Graphics to draw an image directly to the screen .

An Objective-C has only two restrictions related to its implementation of `runWithInput:fromAction:error:`. It cannot return until it has completely finished whatever processing it has initiated. For instance, if an action instructs a camera to take a picture (an asynchronous process) it cannot return from `runWithInput:fromAction:error:` method until the picture is taken. So the action has to implement whatever blocking algorithm, timeout logic, or threading strategy is necessary until the picture is taken. The second restriction has to do with Automator's threading architecture. Because Objective-C actions run on a secondary thread, if they want to display a window it must be done on the main thread. The `performSelectorOnMainThread:withObject:waitUntilDone:` method is adequate for this purpose; for example:

```
self performSelectorOnMainThread:@selector(showPreviewWithData:) withObject:data
 waitUntilDone:YES]
```

Updating Action Parameters

A typical action uses the bindings technology of Cocoa to synchronize the settings on an action's user interface with the action's parameters attribute. (See "Constructing the User Interface" (page 31) in "Developing an Action" (page 29).) But an action is not required to use bindings to do this synchronization. It can choose to do it manually by implementing the `updateParameters` and `parametersUpdated` methods. These methods are invoked went it is time to run or save the action.

The `updateParameters` method is invoked when the parameters attribute of the action (an NSDictionary object) needs to be refreshed from the settings and values the user has specified in the user interface. The `parametersUpdated` method is invoked for the opposite reason: to make the pop-up items, buttons, text fields, and other objects in the action's view reflect the current contents of the action's parameters, especially when those contents change programmatically.

Listing 4 shows a typical implementation of the `updateParameters` method.

Listing 4 Updating an Objective-C action's parameters manually

```
- (void)updateParameters
{
    // text
    NSString *outputFile = [_outputFilePath stringValue];
    if (outputFile)
    {
        [[self parameters] setObject:outputFile forKey:@"outputFilePath"];
    }

    NSString *waitSeconds = [_waitSeconds stringValue];
    if (waitSeconds)
    {
        [[self parameters] setObject:[NSNumber numberWithInt:
            [_waitSeconds intValue]] forKey:@"waitSeconds"];
    }

    // radios
    [[self parameters] setObject:[NSNumber numberWithInt:
        [_interactiveType selectedRow]] forKey:@"interactiveType"];
    [[self parameters] setObject:[NSNumber numberWithInt:
        [_screenShotType indexOfSelectedItem]] forKey:@"screenshotType"];

    // popup
    [[self parameters] setObject:[NSNumber numberWithInt:
        [_outputType indexOfSelectedItem]] forKey:@"outputType"];

    // checks
    [[self parameters] setObject:[NSNumber numberWithBool:
        [_captureMainMonitorOnly state]] forKey:@"captureMainMonitorOnly"];
    [[self parameters] setObject:[NSNumber numberWithBool:
        [_disableSounds state]] forKey:@"disableSounds"];
    [[self parameters] setObject:[NSNumber numberWithBool:
        [_timedScreenshot state]] forKey:@"timedScreenshot"];
}
```

You can also update an action's parameters to include miscellaneous data that can stay with the action as it is, for example, copied to the pasteboard or written to disk in a workflow. When Automator archives an action it includes the action's parameters with the archive, and thus objects stored in the parameters can be unarchived and retrieved. The only restriction is that the Foundation object saved to an action's parameters must be a property-list object—that is, an NSData, NSDate, NSNumber, NSString, NSArray, or NSDictionary object. Any object that is not one of these must be converted to a property-list object before it can be archived.

Objects that represent URLs—in other words, instances of NSURL—are an important case in point. URLs are a type of data (identified with UTI `public.url`) that actions occasionally deal with. Fortunately, there are APIs to convert NSURL objects to and from a property-list object. The code in Listing 5 converts an NSURL object to an NSString object and stores that in the action parameters.

Listing 5 Saving an NSURL object to parameters as a string object

```
NSMutableDictionary* params;

if (mFilterURL) // NSURL object
{
    NSString* path = [mFilterURL path];
    params = [NSMutableDictionary dictionaryWithObject:path  forKey:@"URL ];

    [self setParameters:params];
}
```

Now when the action is archived the URL is stored with it, and when it is unarchived the URL is retrievable. The code fragment in Listing 6 restores the NSURL object from the NSString object stored in the action parameters.

Listing 6 Restoring an NSURL object from an action's parameters

```
NSURL* filterURL = nil;

NSMutableDictionary* params = [self parameters];

NSString* path = [params objectForKey: @"URL"];

if (path) filterURL = [NSURL fileURLWithPath:path];

if (filterURL)
{
   // do something with filterURL here
}
```

This example uses `fileURLWithPath:` to create an NSURL object representing a file-scheme URL. If you want a URL object with a different scheme, you would use something such as `URLWithString:`, an NSURL class method. However, the string for this method must contain any necessary percent-escape codes. The easiest way to obtain such a string to send `absoluteString` to the NSURL object before storing the URL string in the action parameters. If your action for some reason does not deal with absolute URLs, then it must devise a way to save and restore the various parts of the URL.

Loading and Executing Scripts

Objective-C actions can also include AppleScript code that they load and execute, making them in effect hybrid actions. The OSA framework and the Foundation framework contain a number of classes (OSAScript, NSAppleScript, NSAppleEventDescriptor, and more) that enable creation, preparation, and execution of objects representing AppleScript scripts.

Listing 7 illustrates how an action gets an AppleScript script either from its user interface or its parameters, creates an OSAScript object with it, and then compiles and executes the script.

Listing 7 Executing an AppleScript script in an Objective-C action

```
- (OSAScript *)script
{
    OSAScript *script = nil;
    NSString *source = nil;

    if ([self controller])
    {
        [[self controller] compileScript:self];
        source = [[[self controller] scriptView] string];
    }
    else
    {
        source = [self source];
    }

    if (source)
    {
        script = [[OSAScript alloc] initWithSource:source language:[OSALanguage
defaultLanguage]];
        if (script)
        {
            NSDictionary *errorInfo;
            [script compileAndReturnError:&errorInfo];
        }
    }
    return [script autorelease];
}

- (NSString *)source
{
    return [[self parameters] objectForKey:@"RunScriptSource"];
}

- (void)setSource:(NSString *)source
{
    [[self parameters] setObject:source forKey:@"RunScriptSource"];
}
```

Creating a Conversion Action

A conversion action acts as a kind of bridge between two actions whose types of provided data (`AMProvides` property) and accepted data (`AMAccepts`) do not match. The conversion action converts between one type and another, usually from an internally defined data type (such as an iTunes track object, specified by the UTI identifier `com.apple.itunes.track-object`) to an externally defined public type (such as a file, specified by `public.item`).

Automator does not display conversion actions and users do not have to bother placing them between actions. The application determines if there is a data-type mismatch between two actions and, if a suitable conversion action is available, it inserts it invisibly between them. Conversion actions have a bundle extension of `.caction` and are installed in the usual system directories for actions.

You create a conversion action just as you would a "normal" action (as described in "Developing an Action" (page 29)) but with just a few differences:

- Set the extension of the produced bundle to `.caction`. To do this, select the action target and choose Get Info from the Project menu. In the Build pane of the Info window (Customized Settings collection), set the Wrapper Extension to ".caction".
- In the information property list (`Info.plist`) for the bundle be sure to do the following:
  - The `AMAccepts` type identifier should specify the type of data converted *from*.
  - The `AMProvides` type identifier should specify the type of data converted *to*.
  - The `AMCategory` property should have a value of "Converter/Filter".
  - The `AMApplication` property value should be "Automator".

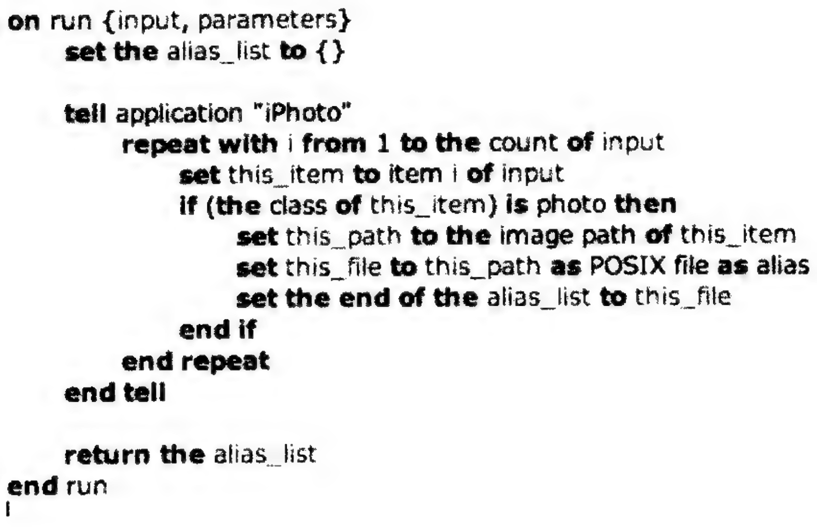
  See Listing 1 for an example.
- Of course, there is no need for an action description, nib file, or similar resource.

Listing 1 Typical Automator properties for conversion actions

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple Computer//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>AMAccepts</key>
    <dict>
        <key>Container</key>
        <string>List</string>
        <key>Types</key>
        <array>
            <string>com.apple.iphoto.photo-object</string>
        </array>
    </dict>
    <key>AMApplication</key>
    <string>Automator</string>
    <key>AMCategory</key>
    <string>Converter/Filter</string>
    <key>AMDefaultParameters</key>
    <dict/>
    <key>AMIconName</key>
    <string>(* The name of the icon *)</string>
    <key>AMName</key>
    <string>Convert Photo object to Alias object</string>
    <key>AMProvides</key>
    <dict>
        <key>Container</key>
        <string>List</string>
        <key>Types</key>
        <array>
            <string>public.item</string>
        </array>
    </dict>
    <!- other properties here -->
</dict>
</plist>
```

As the final step, write the script or Objective-C source code to do the conversion. The script example in Figure 1 converts iPhoto objects representing photo images to paths to those images in the file system.

Figure 1 A typical conversion script



```
on run {input, parameters}
    set the alias_list to {}

    tell application "iPhoto"
        repeat with i from 1 to the count of input
            set this_item to item i of input
            if (the class of this_item) is photo then
                set this_path to the image path of this_item
                set this_file to this_path as POSIX file as alias
                set the end of the alias_list to this_file
            end if
        end repeat
    end tell

    return the alias_list
end run
```

Automator Action Property Reference

When you develop an action for Automator, one of the steps is specifying the properties of the action in its information property list (see "Specifying Action Properties" (page 36)). Automator uses these properties for various purposes, among them determining the types of data the action can accept and produce, setting the initial parameters on the action's user interface, and getting the action's name, icon, associated application, and category. The following sections document the Automator properties and describe the UTI-based type identifiers that you use to specify the types of data an action provides and accepts.

Property Keys and Values

You must specify some of the Automator properties described below in an action project's information property list (`Info.plist`), and optionally may specify the others. The `Info.plist` file in the project template includes, with the exception of `AMRequiredResources`, the Automator properties (with comments for values).

`AMAccepts`

A dictionary that describes the data that the action will accept. This required property has three subproperties:

- `Container`—A string that specifies how multiple inputs are passed. Currently this should have the default value (`List`).
- `Optional`—A Boolean that indicates whether input is optional for this action. If your action can function properly without input, specify `<true/>`; if it requires input, specify `<false/>`. If this property is not specified, Automator assumes a default of `</false>`.
- `Types`—An array of type identifiers indicating the types of data that your action can accept as valid input.

Example:

```
<key>AMAccepts</key>
    <dict>
        <key>Container</key>
        <string>List</string>
        <key>Optional</key>
        <true/>
        <key>Types</key>
        <array>
            <string>com.apple.textedit.document-object</string>
            <string>public.text</string>
            <string>public.item</string>
        </array>
    </dict>
```

For information on valid type identifiers, refer to "Type Identifiers" (page 75). See also the corresponding property `AMProvides`.

`AMApplication`

A string that identifies the primary application that the action uses to do its work. For example, if your action controls the Finder application with AppleScript commands, you would specify "Finder" for this property. Automator uses this property as a filter and as a search criterion. Examples of possible values are Address Book, iCal, and Xcode.

Example:

```
<key>AMApplication</key>
<string>Finder</string>
```

`AMCanShowSelectedItemsWhenRun`

A Boolean that controls whether the action allows the presentation of selected parts of its user interface for the Show When Run feature. If this property is set to `<true/>` (the default), the parts of the action's view (either automatically or manually generated) appear as items in the table view exposed under the Options button. If it is set to `<false/>`, only the "Show Action When Run"checkbox is presented (assuming that the `AMCanShowWhenRun` property is set to `<true/>`).

`AMCanShowWhenRun`

A Boolean that controls whether the Show When Run feature is enabled for the action. By default, it is set to `<true/>`, causing the Options button to be displayed in the lower-left corner of the action view. When the button is clicked, the view is extended to expose a "Show Action When Run" checkbox and, optionally, controls for selecting parts of the action view to present to work flow users (see the description of `AMCanShowSelectedItemsWhenRun`). Setting the `AMCanShowWhenRun` to `<false/>` removes the Options button from the action view.

`AMCategory`

A string that Automator uses to group the action with similar actions in terms of their effects or the objects they operate on. Automator presents categories in its user interface and also uses the category as a search criterion. You can use one of the existing Automator categories or specify a new category. Examples of category names are Find, Music, Text, and Pictures.

Example:

```
<key>AMCategory</key>
<string>Pictures</string>
```

Note: Currently, Automator does not show categories. They are, however, used in searches.

`AMDefaultParameters`

A dictionary containing default values for the user-interface elements managed by Cocoa bindings. The keys are the names of the user-interface elements entered as attributes of the Parameters instance (NSObjectController) in Interface Builder. The values should be contained in elements identifying the type of data; boolean is used for check boxes, integer for pop-up item index, and string for text fields. This property is required for actions that have parameters.

Example:

```
<key>AMDefaultParameters</key>
<dict>
    <key>fromChoice</key>
    <integer>1</integer>
    <key>replaceExisting</key>
    <boolean>No</boolean>
    <key>toChoice</key>
    <integer>0</integer>
    <key>toDirectory</key>
    <string></string>
</dict>
```

`AMDescription`

A dictionary that specifies the content of an action description, which appears in the lower-left view of Automator when the action is selected. The description's icon and title are derived from the `AMIconName` and `AMName` properties, respectively. The `AMDescription` dictionary has the following keys (the values for which are all strings):

- `AMDSummary`—A succinct description of what the action does. It appears directly under the action title. This text is required.
- `AMDInput`—The types of data the action accepts. If this subproperty is not specified, Automator uses default text corresponding to the type identifiers specified for `AMAccepts`. he subheading "Input:" precedes this text in the displayed description.
- `AMDResult`—The types of data the action provides. If this subproperty is not specified, Automator uses default text corresponding to the type identifiers specified for `AMProvides`. The subheading "Result:" precedes this text in the displayed description.
- `AMDOptions`—Describes what can be set in the action's user interface. The subheading "Options:" precedes this text in the displayed description. Use this subproperty only for explaining aspects of the action's user interface that aren't readily apparent.
- `AMDAlert`—Points out a important possible consequence of the action, such as overwriting a file. The subheading "Alert:" precedes this text in the displayed description.
- `AMDNote`—Additional information that the user should know (but not as critical as an alert). The subheading "Note:" precedes this text in the displayed description.
- `AMDRequires`—Anything this action requires in order to work properly, such a web page displayed or a local printer connected. The subheading "Requires:" precedes this text in the displayed description.

- AMDRelatedActions—Specify the bundle IDs of actions that are especially related to this action. (You can look at the Info.plist file inside an action bundle to find out its bundle ID.) For example, a Send iChat Message action is strongly related to a Send Mail Message action.

  Example:

```
<key>AMDescription</key>
    <dict>
        <key>AMDInput</key>
        <string>Project file from the previous action.</string>
        <key>AMDOptions</key>
        <string>Set, retrieve, increment, tag, submit, get
            marketing version, set marketing version. </string>
        <key>AMDResult</key>
        <string>Pass information to next action or to a log
            file.</string>
        <key>AMDSummary</key>
        <string>This action changes the version information of
            an Xcode project. It uses the agvtool shell tool, and
            allows you to set most all of the options of that
            tool.</string>
    </dict>
```

AMIconName

A string that specifies the icon (minus the icon-file extension) that Automator displays to the left of the action name in its user interface. This name can refer to an icon in Automator, an icon in /System/Library/CoreServices/CoreTypes.bundle, or name of a custom icon in the action bundle.

Example:

```
<key>AMIconName</key>
<string>Address Book</string>
```

AMKeywords

An array of strings that give Automator keywords for identifying the action in searches. Keywords are not shown in the Automator user interface but are used in searches. Note that Automator automatically uses words in the names of actions as keywords, so there is no reason to repeat them in this property. Examples keywords are Change, Folder, and Label.

Example:

```
<key>AMKeywords</key>
    <array>
        <string>Add</string>
        <string>Spotlight</string>
        <string>File</string>
    </array>
```

`AMName`

A string giving the name of the action to be displayed in the Automator user interface. This property is required. Example names are "Add Attachments to Front Message", "Copy Files", and "Profile Executable".

Example:

```
<key>AMName</key>
<string>Copy Files</string>
```

See the guidelines for naming actions in "Design Guidelines for Actions" (page 23).

`AMProvides`

A dictionary that describes the data that the action will generate as output. This required property has two subproperties:

- `Container`—A string that specifies how multiple outputs are passed. Currently this should have the default value (`List`).
- `Types`—An array of type identifiers indicating the types of data that your action can provide as valid input.

Example:

```
key>AMProvides</key>
    <dict>
        <key>Container</key>
        <string>List</string>
        <key>Types</key>
        <array>
            <string>public.item</string>
        </array>
    </dict>
```

For information on valid type identifiers, refer to "Type Identifiers" (page 75). See also the corresponding property `AMAccepts`.

`AMRequiredResources`

An array of dictionaries that describes the applications, files, or other resources that the action requires to work properly. If the specified resource is not available when the bundle is loaded, Automator warns the user. If the resource is not available when the action's workflow executes, Automator asks the user locate the missing resource. If the user does not find the resource, the workflow stops executing. This property has four subproperties for each dictionary:

- `Display Name`—A string that identifies the name of the resource as displayed in Finder, for example "iPhoto". Leave this value blank if the `Type` key is set to "file".
- `Resource`—A string identifying the required resource. The format of the resource is indicated by the `Type` subproperty (see note below).
- `Type`—A string indicating the type of the resource identifier given as the value of `Resource`: "application", "creator code", or "file".
- `Version`—A string specifying the minimum version of the resource required by the stage in *n.n.n* format (for example, "4.0.1").

Note: The value for the Resource dictionary key depends on the value for Type. If the type is "application", the resource is specified by the application's bundle identifier; if the type is "creator code", the resource is specified by a four-character creator code; if the type is "file", the resource is specified by full file-system path.

Examples:

```
<key>AMRequiredResources</key>
    <array>
        <dict>
            <key>Display Name</key>
            <string>iTunes</string>
            <key>Resource</key>
            <string>com.apple.iTunes</string>
            <key>Type</key>
            <string>application</string>
            <key>Version</key>
            <string>4.6</string>
        </dict>

        <dict>
            <key>Display Name</key>
            <string></string>
            <key>Resource</key>
            <string>/usr/bin/otool</string>
            <key>Type</key>
            <string>file</string>
            <key>Version</key>
            <string>10.3.5</string>
        </dict>
    </array>
```

AMWarning

If the action can potentially lose data, specify in this dictionary a set of subproperties that cause Automator to display a warning when the action begins executing. The information in the warning comes from the AMWarning subproperties. Optionally, this property can allow the user to insert another action before this one to protect against data loss. A common warning in Automator adds the Copy Files action before the current action. As a result, the action alters copies of files rather than the originals. This property is required but only the value for the Level key needs to be specified (and is set to a default of 0 in the project templates).

- Action—A string that specifies the bundle identifier of an action to be inserted (if the user agrees) in the workflow before this action. Leave this value blank if you just want to warn the user. If you do not specify a value for this key, the ApplyButton string is used for the button title but the IgnoreButton value is ignored.
- ApplyButton—A string that is the title for the button that adds the proposed action or, if no action is specified, that continues the workflow. Examples are "Add" and "Continue".
- IgnoreButton—A string that is the title for the button for rejecting the proposed action or, if no action is specified, for cancelling the workflow. Examples are "Don't Add" and "Cancel".
- Level—An integer that specifies the level of the warning:

  0: Safe operation (default)

1: Makes reversible changes, for example Flip Images

2: Makes irreversible changes, for example Crop Images

This is the only required value of the AMWarning dictionary. If you specify level 1 or 2 you must also specify a Message string.

- Message—A string that is the warning presented to users in the sheet.

The following example shows an AMWarning property that simply warns the user without offering a prior action:

```
<key>AMWarning</key>
    <dict>
        <key>Action</key>
        <string></string>
        <key>ApplyButton</key>
        <string>Continue</string>
        <key>IgnoreButton</key>
        <string>Cancel</string>
        <key>Level</key>
        <integer>2</integer>
        <key>Message</key>
        <string>This action will update the connected iPod. It cannot be
undone.</string>
```

In this example, the property proposes a prior action:

```
<key>AMWarning</key>
    <dict>
        <key>Action</key>
        <string>com.apple.Automator.CopyFiles</string>
        <key>ApplyButton</key>
        <string>Add</string>
        <key>IgnoreButton</key>
        <string>Don't Add</string>
        <key>Level</key>
        <integer>2</integer>
        <key>Message</key>
        <string>This action will change the image files passed into it. Would
you like to add a Copy Files action so that the originals are preserved?</string>
```

Type Identifiers

The Automator AMAccepts and AMProvides properties use type identifiers to specify the types of data for action inputs and outputs. Automator uses the text in the Description column in Table 1 and Table 3 in its user interface to designate the types of input data and output data for the actions in a workflow.

Note: Although the type identifiers described below are based on Uniform Type Identifiers (or UTIs), you can use any string as an identifier. However, such private identifiers have severely limited usefulness in a workflow because actions from Apple and other sources cannot have any knowledge of them. You can, however, create conversion actions for converting the data indicated by the private identifier to and from data indicated by public type identifiers.

Automator actions can use public type identifiers in three categories. The first is specific to the technologies or applications of Apple. Table 1 lists the current identifiers in this category.

Table 1 Type identifiers internally defined by Apple

Type Identifier	**Description**
`com.apple.applescript.object`	Object
`com.apple.applescript.data-object`	Data
`com.apple.applescript.url-object`	URL
`com.apple.applescript.text-object`	Text
`com.apple.finder.file-or-folder-object`	Finder file or folder
`com.apple.idvd.menu-object`	iDVD menu
`com.apple.idvd.slideshow-object`	iDVD slideshow
`com.apple.iphoto.item-object`	iPhoto item
`com.apple.iphoto.album-object`	iPhoto album
`com.apple.iphoto.photo-object`	iPhoto photo
`com.apple.itunes.item-object`	iTunes item
`com.apple.itunes.source-object`	iTunes source
`com.apple.itunes.playlist-object`	iTunes playlist
`com.apple.itunes.track-object`	iTunes track (song)
`com.apple.safari.document-object`	Safari document
`com.apple.textedit.document-object`	TextEdit document
`com.apple.mail.item-object`	Mail item
`com.apple.mail.message-object`	Mail message
`com.apple.mail.mailbox-object`	Mail mailbox
`com.apple.mail.account-object`	Mail account
`com.apple.addressbook.item-object`	Address Book item

Type Identifier	Description
`com.apple.addressbook.group-object`	Address Book group
`com.apple.addressbook.person-object`	Address Book person
`com.apple.ical.item-object`	iCal item
`com.apple.ical.event-object`	iCal event
`com.apple.ical.calendar-object`	iCal calendar
`com.apple.ical.todo-object`	iCal to do

A special subset of the type identifiers defined by Apple are those reserved for Objective-C actions, which are described in Table 2.

Table 2 Objective-C type identifiers

Type Identifier	Description	Comments
`com.apple.cocoa.path`	File	An NSString object representing a POSIX-style path.
`com.apple.cocoa.url`	URL	An NSURL object representing a URL
`com.apple.cocoa.string`	Text	An NSString object.

The final category of type identifier includes externally defined UTIs. Table 3 lists these public identifiers.

Table 3 Externally defined type identifiers

Type Identifier	Comments
`public.data`	Data
`public.url`	URL
`public.text`	Text
`public.image`	Image file
`public.item`	File/Folder
`com.apple.quicktime-movie`	QuickTime movie

You can convert between type identifiers—and most importantly between externally and internally defined identifiers—using a special kind of action called a conversion action. See "Creating a Conversion Action" (page 67) for details.

Document Revision History

This table describes the changes to *Automator Programming Guide*.

Date	Notes
Tiger	New document that explains how to create actions--loadable bundles that perform discrete tasks--for the Automator application.

Index